UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Sabrine Piovesana

TUNING EM SQL *SERVER*
ESTUDO DE CASO

Caxias do Sul
2010

Sabrine Piovesana

TUNING EM SQL *SERVER*
ESTUDO DE CASO

Trabalho de Conclusão de Curso para obtenção do Grau de Bacharel em Ciência da Computação da Universidade de Caxias do Sul.

Daniel Luís Notari
Orientador

Caxias do Sul
2010

Aos meus pais, Alcides e Teresinha,
e meu irmão Bruno pelo amor,
dedicação, exemplo e incentivo
a tudo que realizo.

AGRADECIMENTOS

Agradeço a Deus por estar comigo durante esses longos anos de estudo.

À meus pais, Alcides e Teresinha, que sempre estiveram ao meu lado nos momentos mais difíceis, sempre me apoiando com palavras de força e coragem, que não me fizeram desistir. Vocês são um exemplo de vida e dedicação aos filhos. Amo vocês, obrigada por tudo.

À meu namorado, Bruno, pelo amor, carinho, atenção, ajuda e compreensão nesses longos meses de trabalho.

Ao meu orientador, Daniel, pela atenção e dedicação durante o semestre, pelas palavras de tranquilidade nos momentos mais inseguros.

Aos diretores da HOS Sistemas, pelas oportunidades dadas e pela compreensão das minhas faltas, ao longo da vida acadêmica.

Aos meus amigos, que sempre souberam compreender minhas ausências. Em especial, às amigas Ana Paula, Camila e Priscila pela amizade e companheirismo nesses sete anos dedicado a vida acadêmica, pelos momentos de dificuldade e alegria que passamos juntas. Somos vencedoras!

RESUMO

Este trabalho apresenta um estudo sobre as técnicas de *tuning* com o objetivo de melhorar o desempenho de consultas em uma aplicação no SGBD SQL *Server* 2008. São apresentados os princípios básicos de *tuning* e suas abordagens, ressaltando a importância da aplicação do *tuning* desde o início do projeto, com a utilização de uma metodologia. Uma das técnicas de *tuning* é o uso eficiente dos índices. Estes devem ser utilizados com conhecimento e responsabilidade para que não causem impactos negativos ao sistema. Outra técnica é analisar os tipos de junções, onde suas utilizações variam de acordo com a modelagem dos dados, tamanho das tabelas envolvidas nas junções e a presença, ou não, de índices nos atributos que são utilizados como ligação entre as tabelas. As ordenações e agrupamentos, presentes na maioria das consultas de um aplicação, também foram destacadas. E por fim, alguns *hints,* disponíveis no SGBD SQL *Server*, são conceitualizados e exemplificados, a fim de melhorar as consultas quando o otimizador não consegue realizá-las de forma eficiente ou quando necessita-se utilizar algum recurso específico. Estas técnicas foram aplicadas em estudos de casos no SQL *Server* 2008 que envolveu a remodelação do módulo de vendas de uma aplicação comercial.

Palavras-chaves: *tuning* de consulta, banco de dados, desempenho, índice.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Sigla	Significado em Português	Significado em Inglês
DBA	Administrador de Banco de Dados	*DataBase Administrator*
DBF	Arquivo de Banco de Dados	*DataBase File*
DER	Diagrama de Entidade Relacionamento	*Entity Relationship Model*
DML	Linguagem de Manipulação de Dados	*Data Manipulation Language*
DOS	Sistema Operacional em Disco	*Disk Operating System*
FK	Chave Estrangeira	*Foreign Key*
PK	Chave Primária	*Prmary Key*
RAM	Memória de Acesso Aleatório	*Random Access Memory*
SGBD	Sistema Gerenciador de Banco de Dados	*DataBase Management System*
SQL	Linguagem de Consulta Estruturada	*Structured Query Language*
XML	Linguagem de Marcação Extensível	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Com o aumento da utilização de sistemas para gerenciamento de informações em organizações, de pequeno a grande porte, onde o armazenamento de dados deve ocorrer de forma eficiente e segura, exige-se meios para obtenção de resultados rápidos e consistentes.

Para suprir essa necessidade, utiliza-se o SGBD, que tem como principal finalidade armazenar de forma organizada os dados, visando à otimização dos sistemas e o uso adequado das informações (ELMASRI & NAVATHE, 2005).

Cada vez mais a necessidade de armazenamento de grandes volumes de dados e a importância dessas informações para tomada de decisões nas organizações, tem exigido dos bancos de dados um alto desempenho no que se refere à velocidade de retorno dos dados. Sendo assim, um dos fatores principais utilizados para medir a qualidade de uma aplicação é justamente seu desempenho mediante as operações realizadas. Os usuários necessitam das informações o mais breve possível, por isso não podem ficar à mercê do SGBD, aguardando que o processamento se concretize para que ele possa prosseguir com suas tarefas.

Problemas de desempenho de SGBD ocorrem quando determinadas operações de consultas levam mais tempo que o esperado para serem executadas. Vários aspectos do projeto de sistema de banco de dados afetam o desempenho de uma aplicação, variando desde aspectos de alto nível até tópicos de hardware. Dessa forma torna-se necessário o uso de técnicas e processos que possam melhorar o tempo de resposta das consultas. Neste contexto, evidencia-se o *tuning* como um processo de melhoria aplicável ao sistema como um todo, passando por melhorias que podem atingir o *hardware*, projeto de banco de dados e consultas (SILBERSCHATZ et. al.,1999).

Porém, não há regras definidas para aplicação de *tuning*. Deve-se sempre avaliar, de forma criteriosa, quais os gargalos da aplicação, encontrando as soluções aplicáveis em cada caso. O processo de *tuning* de banco de dados não é uma tarefa para ser executada apenas uma vez, deve ser sempre aprimorado de acordo com os novos cenários que são apresentados diariamente, conforme destaca Elmasri & Navathe (2005),“o *tuning* de banco de dados continua enquanto o banco de dados existir, enquanto os problemas de desempenho forem descobertos e enquanto os requisitos continuarem a se modificar”.

1.1 OBJETIVOS DO TRABALHO

Este trabalho tem como principal objetivo apresentar as técnicas de *tuning*, abordando, principalmente, maneiras de otimizar um comando de consulta SQL, quando esse apresentar um desempenho inadequado, utilizando as seguintes técnicas:

- Conhecer e aplicar corretamente os índices;
- Conhecer e aplicar os diferentes métodos de junções;
- Otimizar as consultas que contenham ordenações e agrupamentos;
- Utilizar os *hints*, disponíveis no SQL *Server* 2008, de forma adequada.

1.2 ORGANIZAÇÃO DO TRABALHO

No capítulo 2 são apresentados os conceitos de *tuning*, bem como seus princípios básicos, destacando as abordagens, os tipos de *tuning* e uma metodologia que pode ser utilizada para aplicação dessa técnica.

No capítulo 3 é abordado o *tuning* de consulta, com ênfase nos tipos de índices e sua utilização de modo eficiente, nos métodos de junções utilizados para compor resultados a partir da ligação de duas ou mais tabelas, nas formas de otimização de consultas que contêm ordenações e agrupamentos, e finalizando, no uso de *hints*.

No capítulo 4 é apresentado a proposta de solução criada, destacando o cenário o atual do banco de dados, que será utilizado para aplicação das técnicas de *tuning*, bem como o projeto de banco de dados e de *tuning* de consulta aplicado.

No capítulo 5 é apresentada a ferramenta de importação de dados desenvolvida para importar os dados da base de dados atual para a base de dados criada, conforme estrutura apresentada nos Anexos E e F.

No capítulo 6 são apresentados os sete estudos de caso realizados, aplicando as técnicas de *tuning* de consulta para agrupamento, ordenação, uso efetivo de índices e aplicação de *hints*. Em cada estudo de caso, são apresentadas as regras de negócio e o problema de cada consulta, além de destacar o plano de execução e os gráficos comparativos entre as consultas com e sem aplicação das técnicas de *tuning*.

2 *TUNING* DE BANCO DE DADOS

As organizações dependem, cada vez mais, que os sistemas de banco de dados estejam disponíveis e, operando de forma eficiente, para que possam atender a demanda de suas operações. Para garantir um melhor desempenho e estabilidade, estão investindo tempo e dinheiro para realizar uma revisão e otimização do sistema de banco de dados.

Porém, o termo desempenho é subjetivo quando se trata de consultas em banco de dados, pois enquanto um usuário está satisfeito com uma consulta que leva dez segundos para executar, outro usuário pode não se satisfazer a menos que a consulta leve um segundo (SACK, 2005).

Para melhorar a performance das consultas, a técnica conhecida como *tuning* ou sintonia, pode ser aplicada nos banco da dados. Essa técnica é definida por Shasha & Bonnet (2003) como "a atividade de fazer uma aplicação de banco de dados executar mais rapidamente. "Mais rapidamente" geralmente significa um maior rendimento, porém, pode significar menor tempo de resposta pelo tempo de aplicações críticas".

Nas próximas seções serão explicados os princípios, as abordagens, os tipos de *tuning* e as metodologias que podem ser aplicadas.

2.1 PRINCÍPIOS BÁSICOS DE *TUNING*

Tuning de banco de dados é uma tarefa que pode ser considerada simples e complexa ao mesmo tempo. Simples pois é livre de modelos matemáticos e, complexa, porque exige conhecimentos profundos da aplicação, do software do banco de dados, do sistema operacional e da estrutura de hardware (SHASHA & BONNET, 2003).

Segundo Fritchey & Dam (2009), o processo de *tuning* consiste em "identificar os gargalos, resolver os problemas de suas causas aplicando diferentes soluções e, em seguida, quantificar se a performance foi melhorada".

Essa atividade envolve cinco princípios básicos que estão diretamente relacionados com a performance do banco de dados. O primeiro deles trata de pensar globalmente no problema, porém realizar os ajustes localmente. Segundo Shasha & Bonnet (2003) "a

aplicação de *tuning* de forma eficiente exige uma adequada identificação do problema e uma intervenção minimalista. Isso implica em ter que medir as quantidades exatas e chegar às conclusões corretas".

Porém, fazer isso é bastante desafiador, pois deve-se entender o funcionamento de todo o sistema, mas os ajustes devem ser realizados em alguns pontos específicos, tomando medidas que ajam efetivamente sobre a causa real do problema. Como exemplo pode-se citar o processamento de uma consulta que esta exigindo alta atividade de disco, isso não significa que o problema esteja no hardware. Ele pode estar relacionado a estrutura de escrita da consulta que varre toda a tabela para retornar o resultado esperado. Esse problema poderia ser resolvido criando um índice na tabela envolvida ao invés de ter que adquirir um hardware (SHASHA, 1992; SHASHA & BONNET, 2003).

O segundo princípio esta relacionado com a detecção de gargalos no sistema, conforme é destacado por Shasha & Bonnet (2003) "um sistema dificilmente está lento porque todos os seus componentes estão saturados. Geralmente, uma parte do sistema acaba limitando sua performance como um todo, e isso é chamado de gargalo". A primeira abordagem que deve ser aplicada para eliminação de gargalos é agilizar o componente que causa o gargalo, ou seja, é uma solução local que corresponde, por exemplo, a reescrever uma consulta para fazer melhor uso de índices existentes. A segunda abordagem trata-se em realizar particionamento, que consiste em reduzir a carga de um certo componente do sistema, dividindo-a entre os componentes ou espalhando-a através do tempo (SHASHA & BONNET, 2003).

O terceiro princípio tem com objetivo obter o resultado esperado com o menor número possível de inicializações, já que o alto custo de processamento está ao iniciar uma operação de leitura, após iniciar a leitura as informações são retornadas rapidamente. Isso sugere que as consultas que geralmente são executadas devem ser compiladas, ou seja, fazendo leituras mais longas e colocando as tabelas que sofrem leitura constantemente serem armazenadas próximas uma das outras (SHASHA & BONNET, 2003).

O quarto princípio trata de uma questão importante que deve ser considerada: a alocação de trabalho entre o sistema de banco de dados (servidor) e o programa de aplicação (cliente). Para que ambos possam trabalhar de forma eficiente, deve-se realizar um balanceamento correto das tarefas para não sobrecarregar o servidor de banco de dados com tarefas que poderiam ser executadas em outras camadas. (SHASHA, 1992; SHASHA &

BONNET, 2003).

O último princípio aborda a importância de saber que, na maioria das vezes, melhorar o desempenho de alguns resultados pode piorar os de outros, por isso é muito importante pesar os prós e contras de cada ação realizada. Segundo Shasha & Bonnet (2003) “aumentar a velocidade de uma aplicação geralmente requer uma combinação de memória, disco ou recursos computacionais”, devido ao fato de estarem amplamente relacionados entre si.

Como exemplo pode-se citar a criação de um índice que, geralmente, faz com que uma consulta considerada crítica torne-se mais eficiente, porém tal ação implica em maior armazenamento de disco e maior espaço de memória. Além disso irá necessitar de mais tempo para processar transações de inserção e atualização (SHASHA & BONNET, 2003).

Os princípios apresentados mostram a importância de pensar em cada decisão a ser tomada quando o assunto é *tuning*, pois essa tarefa envolve um conhecimento geral do sistema para poder detectar os gargalos e corrigi-los de maneira eficiente, sem que isso prejudique outras partes do sistema.

A seguir serão apresentas as abordagens de *tuning*.

2.2 ABORDAGENS DE *TUNING*

Existem duas abordagens que podem ser adotadas para aplicação de *tuning*: a pró-ativa e a reativa. A abordagem pró-ativa geralmente ocorre durante o desenvolvimento, onde envolve a concepção e desenvolvimento da arquitetura de desempenho do sistema, durante os primeiros estágios de uma implementação. Após projetar o banco de dados e escrever as consultas, com base na aplicação e nos requisitos do usuário final, se elas não tiverem um bom desempenho, é possível ajustá-las antes de implantar o sistema (SACK, 2005; VAIDYANATHA, 2001). Tem como principal objetivo evitar problemas críticos antes que eles ocorram e causem danos ao banco de dados. Grande parte da atividade pró-ativa está relacionada com o desempenho do *tuning* do banco de dados de uma forma ou de outra (POWELL, 2007). Através dela, é possível identificar os pontos vulneráveis do sistema, melhorar a produtividade dos recursos e prevenir incidentes.

A Figura 1 ilustra o custo relativo ao *tuning* durante o ciclo de vida de uma aplicação, que é inversamente proporcional ao tempo do projeto, ou seja, quando antes iniciar a aplicação de *tuning* no projeto, menor será o seu custo.

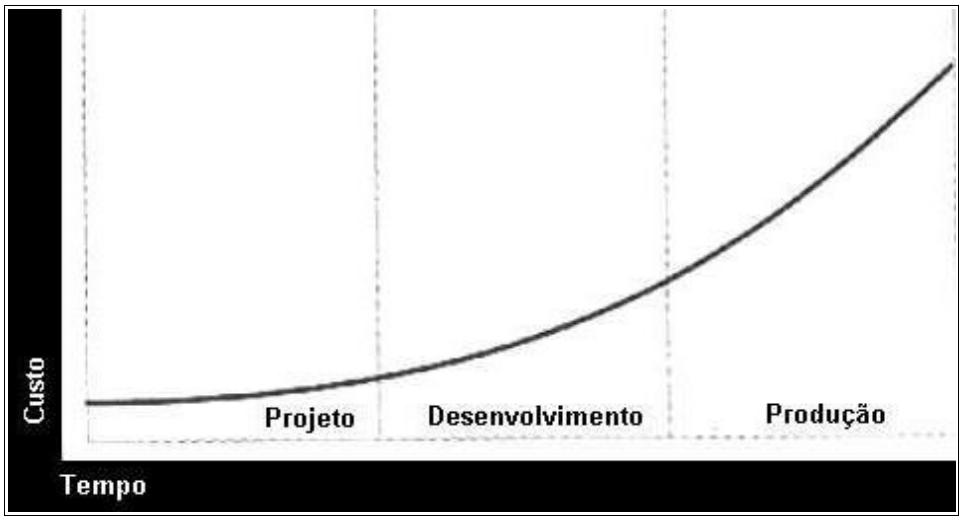


Figura 1: Custo de *tuning* durante o ciclo de vida de uma aplicação
Fonte: ORACLE (2000)

Já a abordagem reativa, envolve a avaliação de desempenho, encontrando soluções para problemas após ter sido realizada a implantação. Essa avaliação envolve ajustes e otimização do ambiente do banco de dados, dentro dos limites de *hardware* existente e desempenho da arquitetura (SACK, 2005; VAIDYANATHA, 2001). Sendo assim, fica evidente que a abordagem pró-ativa é a melhor opção a ser utilizada. Se realmente a fizer de forma correta, a abordagem reativa pode não precisar ser aplicada.

Na próxima seção serão apresentados os tipos de *tuning*.

2.3 TIPOS DE *TUNING*

Segundo Elmasri & Navathe (2005) "após um banco de dados ser desenvolvido, estar em operação, o uso das aplicações, das transações, das consultas e das visões revela fatores e áreas de problemas que podem não ter sido considerados durante o projeto físico inicial".

A fase de *tuning* de banco de dados envolve modificações em vários aspectos, que abordam mudanças nos conceitos aprendidos nos Diagramas Entidade-Relacionamento (DER) até a troca de *hardware*, passando pela estruturação das consultas SQL. Esses aspectos podem então ser divididos em três tipos: *tuning* de *hardware* ; *tuning* de projeto e *tuning* de consulta (SILBERSCHATZ et. al. 2005). Esses três tipos serão explicados nas próximas subseções.

2.3.1 *TUNING* DE *HARDWARE*

A escolha incorreta e a má utilização do *hardware* podem afetar significativamente a performance do servidor que armazena o banco de dados. Os principais componentes que devem ser constantemente avaliados são memória RAM, onde seu aumento evita que o sistema faça paginação de memória para o disco; processador, que com o aumento de velocidade pode melhorar o rendimento e diminuir o tempo de resposta; e disco rígido, onde a velocidade e a configuração podem ter um efeito dramático sobre o desempenho de servidores que executam aplicativos fortemente dependentes de suporte de banco de dados. A arquitetura de rede também é um fator importante que deve ser considerado (BALTER, 2006; IBM, 2004).

Porém, somente a atualização de componentes de *hardware* pode não resolver o problema de performance na aplicação. Deve-se fazer um trabalho conjunto de ajuste do projeto de banco de dados e ajustes de consultas (SHASHA & BONNET, 2003).

2.3.2 *TUNING* DE PROJETO

Conforme Elmasri & Navathe (2005) "se um dado projeto físico de banco de dados não atinge os objetivos esperados, poderemos voltar ao projeto lógico do banco de dados, fazer ajustes no esquema lógico e remapeá-lo em um novo conjunto de tabelas e índices físicos". Algumas das alterações que podem ser realizadas no esquema lógico e no projeto físico do banco de dados são, segundo Elmarsri & Navathe (2005):

- Junção de tabelas existentes, devido a atributos de duas ou mais tabelas serem frequentemente necessários em conjunto.
- Atributos de uma tabela podem ser repetidos em uma outra mesmo que isso crie redundância e um anomalia potencial.
- Realizar particionamento horizontal, obtendo fatias horizontais de uma tabela e as armazenando em tabelas distintas.

Apesar de esses ajustes irem contra alguns princípios de modelagem utilizada em banco relacionais, eles são projetados para atender a consultas e transações de grandes volumes (ELMASRI & NAVATHE, 2005).

2.3.3 *TUNING* DE CONSULTA

A justificativa para se utilizar o *tuning* de consulta como uma das primeiras técnicas é que essa afeta apenas uma consulta específica, não se propagando para outras aplicações que acessam as tabelas envolvidas na instrução SQL. Reescrever uma consulta para ser executada mais rapidamente só oferece benefícios a aplicação (SHASHA & BONNET,2003).

Alguns fatores indicam a necessidade de se realizar esse tipo de *tuning*. Destacam-se entre eles a grande quantidade de acesso a disco quando uma consulta é realizada e a identificação, através do plano de consulta, de que índices relevantes não estão sendo utilizados (ELMASRI & NAVATHE, 2005).

A seguir será apresentado uma metodologia para ajuste de desempenho.

2.4 METODOLOGIA PARA AJUSTE DE DESEMPENHO

Problemas de performance podem estar em qualquer ponto do sistema, por isso uma metodologia completa deve envolver o sistema como um todo. Os passos a seguir fornecem um método recomendado para *tuning* em banco de dados Oracle, que recomenda que os passos sejam seguidos na ordem em que serão apresentados, sendo que os passos com maior impacto devem ser realizados primeiro. A Figura 2 ilustra a metodologia para ajuste de desempenho.

Como pode-se visualizar, o processo de *tuning* é iterativo. Isso faz com que ganhos de performance em um passo possam influenciar em outros. Por exemplo, reescritas de declarações SQL no passo cinco podem ter influências significativas sobre o passo sete. Embora a figura mostre um laço de volta a primeira etapa, pode-se voltar a etapas anterior a qualquer momento.

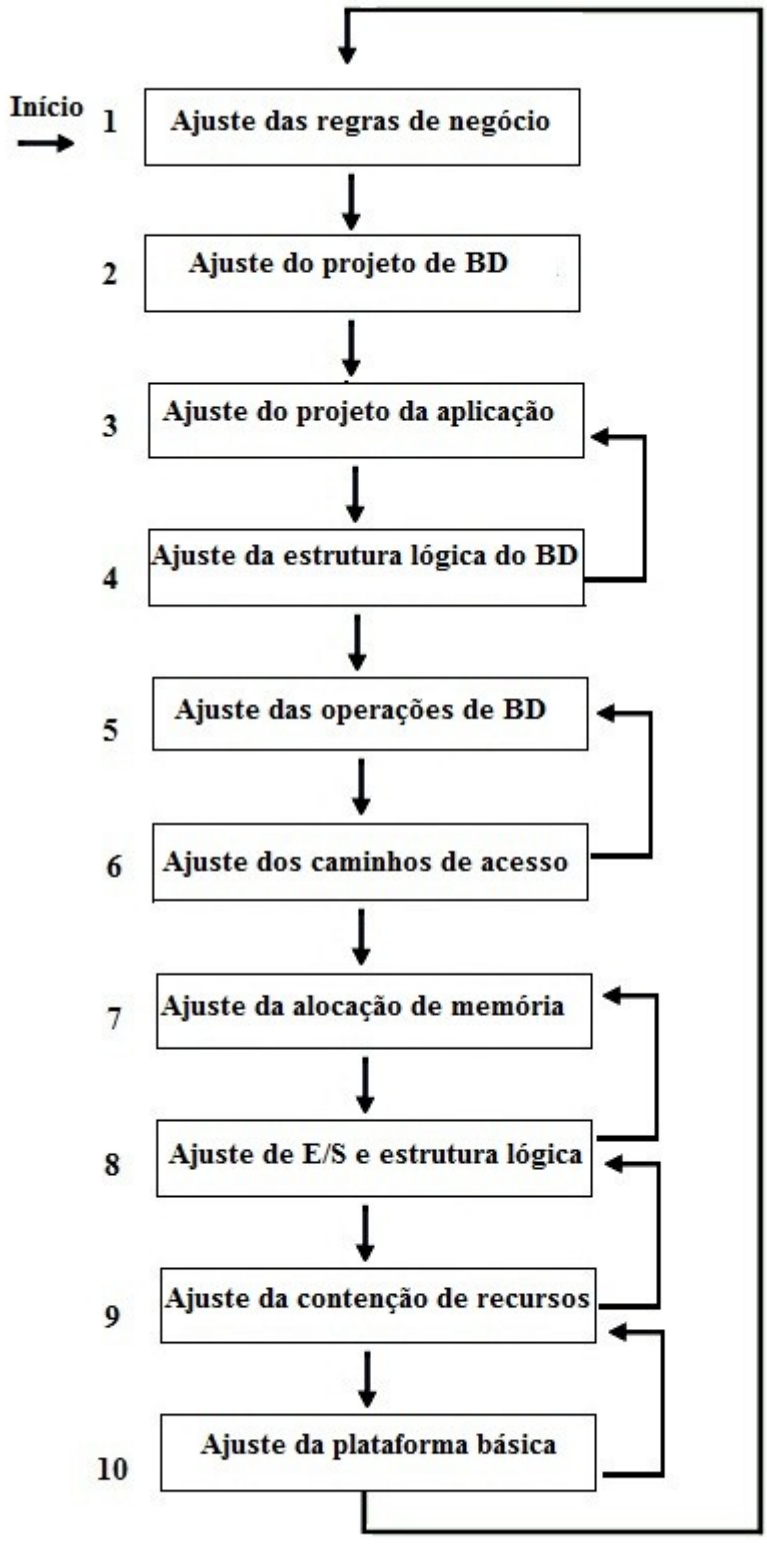


Figura 2: Metodologia de ajuste de desempenho
Fonte: ORACLE (2000)

A seguir serão descritos cada um dos passos sugeridos pela metodologia de ajuste de desempenho (ORACLE, 2000):

- Passo 1 – Ajuste das Regras de Negócio: Para se obter um desempenho ideal, pode-se ter que adaptar as regras de negócio. Isso se refere a uma análise de todo o sistema em mais alto nível. Deste modo, os planejadores garantem que os requisitos de desempenho do sistema correspondam diretamente às necessidades do negócio. Além disso, elas devem ser coerentes com expectativas realistas para a aplicação, como por exemplo, prever o número de usuários simultâneos, o tempo de resposta das transações, o número de registros *on-line* que podem ser armazenados (ORACLE, 2000).

- Passo 2 – Ajuste do Projeto de Banco de Dados: Na fase de projeto de base de dados, deve-se determinar que dados são necessários ao sistema, de acordo com as regras de negócio. Também deve-se definir quais os relacionamentos e atributos são importantes e, consequentemente, as chaves primárias e estrangeiras. Nesse momento também é definido qual o nível de normalização a ser utilizado (ORACLE, 2000).
  Um banco de dados mal projetado levará a um mau desempenho. Segundo Milani (2008) "em um banco de dados normalizado, o espaço em disco necessário para armazenar as informações normalmente é menor, e seu desempenho tende a ser o mais rápido". Porém, apesar de o recomendado ser normalizar uma entidade até a 3º forma normal, o uso excessivo dessa prática pode trazer um desempenho ruim ao sistema. Isso se deve ao fato que os relacionamentos podem não refletir os caminhos de acesso normal que serão empregados para acessar os dados, principalmente em consultas que retornam um grande números de atributos (BRYLA & LONEY, 2009).

- Passo 3 – Ajuste do Projeto da Aplicação: O principal desafio desse passo é transformar objetivos de negócio em um projeto de sistema efetivo. Os processos de negócio referem-se a uma aplicação particular dentro do sistema ou uma parte da aplicação. Um exemplo de projeto de processo é deixar alguns dados em *cache*. Desde modo evita-se ter que recuperar uma mesma informação várias vezes durante o dia (ORACLE, 2000).

- Passo 4 – Ajuste da Estrutura Lógica do Banco de Dados: Após a modelagem do sistema e da aplicação, deve-se planejar a estrutura lógica do banco de dados. Nesta fase deve-se criar índices adicionais, além das chaves primárias e estrangeiras. É necessário que se faça um ajuste fino do projeto de índices, para garantir que não ocorra excessos ou escassez de índices (ORACLE, 2000).

- Passo 5 – Ajuste das Operações de Banco de Dados: O objetivo dessa fase é utilizar todas as vantagens da SQL e as características do sistema de gerenciamento de banco de dados (SGBD) para o processamento da aplicação (ORACLE,2000). As consultas podem ser criadas de forma ineficiente devido a fatores como falta de experiência e conhecimento técnico dos desenvolvedores. Sendo assim, o DBA deve acompanhar o processo de escrita das SQL, orientando os profissionais envolvidos no processo a utilizar corretamente os recursos do banco de dados.

- Passo 6 – Ajuste dos Caminhos de Acesso: Neste passo, deve ser assegurado o acesso eficiente aos dados, minimizando a quantidade de registros a serem pesquisados considerando para isso o uso de *clusters*, *hash clusters*, índices B-*Tree* e índices de *bitmap* (ORACLE, 2000).

  Para visualizar o caminho de acesso aos dados, pode-se utilizar o plano de execução gerado que mostra, dentre outras estatísticas, o caminho de execução da consulta (BRYLA & LONEY, 2009). Através dele é possível identificar os gargalos na realização da consulta na base de dados.

- Passo 7 – Ajuste da Alocação de Memória: Em geral, os SGBDs tem um módulo de armazenamento temporário, conhecido como *cache*, que mantém partes do banco de dados armazenado na memória principal (ELMASRI & NAVATHE, 2005). Por isso, o investimento nesse componente e a alocação apropriada de recursos de memória, para as estruturadas do SGBD, trazem benefícios na performance do *cache*, reduzem a análise de comandos SQL e reduzem a paginação (ORACLE, 2000).

- Passo 8 – Ajuste de Entrada/ Saída (E/S) e Estrutura Física: O processo de *tuning* de

E/S e estruturas físicas envolve a distribuição dos dados de acordo com os dispositivos existentes, evitando contenção de disco (ORACLE, 2000). Esse tipo de *tuning* só trará resultados se o sistema de E/S estiver operando no limite de sua capacidade e, não é mais capaz de atender as solicitações dentro de um prazo aceitável (CHAN, 2008).

- Passo 9 – Ajuste da Contenção de Recursos**:** O processamento concorrente de vários usuários pode criar a contenção de recursos, e isso faz com que os processos aguardem os recursos serem disponibilizados. Um ajuste no processo concorrente, evita a espera dos usuários e melhora o desempenho da aplicação como um todo (ORACLE, 2000).

- Passo 10 – Ajuste das Plataformas Básicas: Problemas de desempenho do sistema operacional geralmente envolvem gerenciamento de processos, gerenciamento de memória e programação. Conforme a versão do gerenciador para um determinado sistema operacional pode haver parâmetros diferentes de ajuste que melhoram ou pioram o desempenho da aplicação, por isso deve-se tomar cuidado ao configurar esses parâmetros (ORACLE, 2000).

Antes de iniciar o processo de *tuning* deve-se estabelecer objetivos claros, de acordo com a aplicação. Esses objetivos devem estar sempre em mente para considerar cada ação de ajuste que deve ser realizada, pesando os prós e contras. Mesmo que uma aplicação tenha todo o seu processo ajustado para garantir um desempenho ideal, ao longo do tempo ela poderá sofrer perda de performance devido a diversos fatores como por exemplo crescimento da base de dados, novos caminhos de acesso e alteração nas regras de negócio.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentado, inicialmente, os princípios básicos para a realização de *tuning* de forma eficiente, descrevendo cada um deles. Após, foi destacada a importância de se realizar esse processo desde o início do projeto da aplicação, garantindo um melhor controle sobre a performance do sistema e reduzindo os custos do projeto. Por fim, foram detalhados os três principais tipos de *tuning*, que estão diretamente relacionados a metodologia de ajuste de desempenho desenvolvida pela Oracle. A metodologia apresentada descreve os dez passos

que devem ser considerados quando decide-se melhorar o desempenho da aplicação como um todo.

No próximo capítulo será abordado o *tuning* de consulta, destacando suas principais aplicações e funcionalidades disponíveis no SQL *Server*, para otimização de consulta.

3 *TUNING* DE CONSULTA

No processo de *tuning* de consulta o objetivo principal é, segundo Bevaart (2006), "executar as consultas existentes com o melhor desempenho possível". Para alcançar esse objetivo pode-se trilhar diversos caminhos. Dentre os mais importantes estão a criação de índices, a recuperação apenas de colunas e linhas necessárias e a utilização de colunas indexadas na escrita do SQL, com isso a velocidade de execução de uma consulta pode ser aumentada enormemente com um custo muito baixo. No entanto, há um limite que pode ser atingindo com o *tuning* de consulta. Quando esse limite for alcançado deve-se partir para recursos extras, como ajustes no *hardware* e no projeto de banco de dados (BEVAART, 2006; FRITCHEY & DAM, 2009).

Umas das principais vantagens da linguagem SQL é não precisar dizer ao banco de dados exatamente como ele deve obter os dados solicitados. Para isso, somente é necessário executar uma consulta especificando as informações desejadas que o *software* de banco de dados encontra a melhor maneira de obtê-las (PRICE, 2009). De acordo com Bryla & Loney (2009) "a chave para ajustar a SQL é minimizar o caminho de pesquisa que o banco de dados usa para localizar os dados".

Assim, nas próximas seções serão destacadas a utilização de índices de modo eficiente, uso de junção, ordenação e agrupamento para melhorar o desempenho de consultas, bem como a utilização de *hints*[1] em junções, consultas e tabelas. Os exemplos de SQL que serão utilizados nesse capítulo baseiam-se no modelo relacional conforme Anexo B.

3.1 ÍNDICES

Segundo Ramalho (2005), índice é "um arquivo auxiliar associado a uma tabela cuja finalidade é acelerar o tempo de acesso às linhas de uma tabela. Um índice é composto por chaves que se baseiam no conteúdo de uma ou mais colunas da tabela". Seu uso em banco de dados é semelhante ao índice remissivo de um livro, onde após procurar o termo desejado no índice, encontra-se facilmente a página desejada (RAMALHO, 2005).

1 São estratégias especificadas para aplicação pelo processador de consultas. Substituem todos os planos de execução que o otimizador de consulta possa selecionar (MSDN, 2009)

De acordo com Milani (2008), a técnica de indexação “talvez seja o método de otimização mais importante, pois é o que consegue obter melhores resultados em menor espaço de tempo”.

A utilização de índices é recomendada quando uma coluna for usada como parâmetro em um dos seguintes argumentos (MSDN, 2009):

- Cláusula *where* de comandos *select* e *update;*
- Ordenação de valores utilizando a cláusula *order by*;
- Junções entre tabelas.

Porém, a utilização de índices tem a desvantagem do tempo adicional gasto para atualizá-lo quando um novo registro for inserido no banco de dados (PRICE, 2009).

Antes de criar um índice, deve- se compreender as características do banco de dados, consultas e colunas de dados, para poder auxiliar a projetar melhores índices (MSDN, 2009). O SQL *Server* possui três tipos principais de índices: clusterizado, não-clusterizado e exclusivo.

Os índices clusterizados organizam as linhas das tabelas na mesma ordem das entradas do índice. Esse tipo de índice é ótimo quando não há muitas operações de atualização de tabela, pois a cada alteração do conteúdo de um campo chave, a tabela é reorganizada. Cada tabela pode ter somente um índice clusterizado (RAMALHO, 2005).

A Figura 3 mostra a estrutura dos índices clusterizados. No SQL *Server*, os índices são organizados como árvores B, sendo que essa árvore é composta por nó raiz, níveis intermediários e nó folha. Esse último nível contêm as páginas de dados da tabela subjacente, enquanto que os nós de nível intermediário e raiz contêm páginas de índices com linhas de índice. Sendo assim, cada linha de índice contém um valor de chave e um ponteiro para uma página de nível intermediário ou uma linha de dados no nó folha do índice (MSDN, 2009; RAMALHO, 2005).

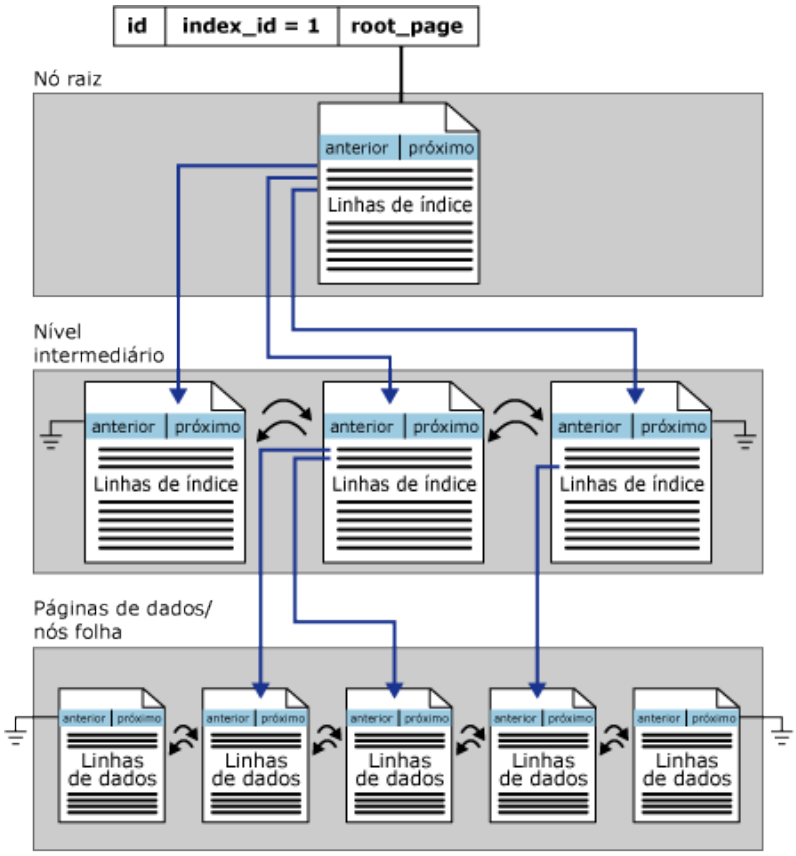


Figura 3: Estrutura do índice clusterizado
Fonte: MSDN (2009)

Já os índices não-clusterizados tem sua estrutura separada da tabela, onde a ordem física das linhas não segue a ordem do arquivo de índice. Esse tipo de índice pode ser comparado ao índice de um livro, em que o índice encontra-se no final do livro e os dados no início, em diferentes seções. Assim, os dados do arquivo de índices são armazenados na ordem de seus valores, já o dado na tabela (registro) é armazenado em outra ordem, que pode não ter a mesma sequência do arquivo de índices (MSDN, 2009; RAMALHO, 2005).

A Figura 4 mostra a estrutura dos índices não-clusterizados. Esse tipo de índice possui a estrutura de árvore B semelhante a do índice clusterizado, porém com duas diferenças: as linhas de dados não são armazenadas na ordem baseada no conteúdo de sua chave e, o nó folha é composto por linhas de índices (MSDN, 2009; RAMALHO, 2005).

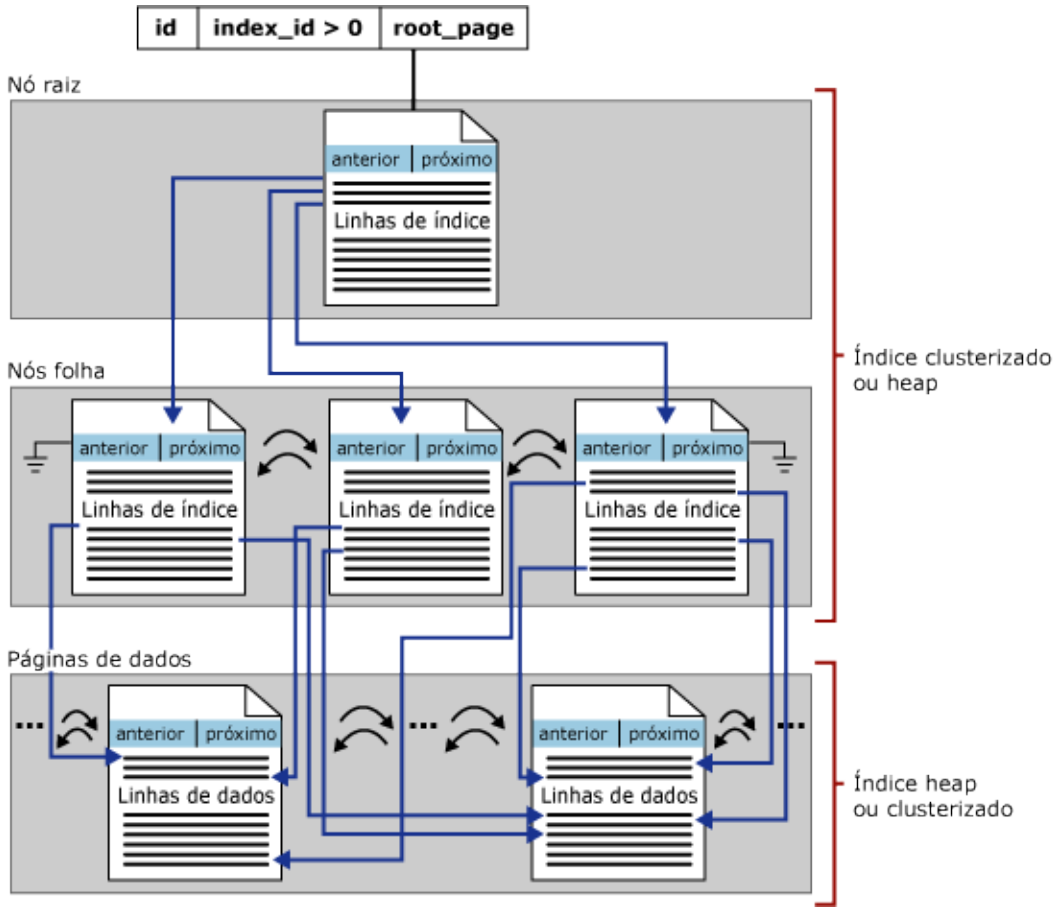


Figura 4: Estrutura do índice não-clusterizado
Fonte: MSDN(2009)

O índice exclusivo garante que a chave de índice não contenha valores duplicados, ou seja, cada linha é única na tabela. Um índice desse tipo só faz sentido quando a exclusividade for uma característica da coluna que esta sendo indexada. Quando é criada uma restrição de chave primária ou única, automaticamente é gerado um índice exclusivo nas colunas especificadas, que também pode ser considerado um índice clusterizado (MSDN, 2009).

O SQL *Server* também possui uma característica conhecida como *fill factor*, que durante a criação de um índice, deixa um espaço livre dentro da página de forma a permitir alterações ou futuras inserções de linhas que possam encontrar espaço dentro da página sem a necessidade de executar o remanejamento das páginas (RAMALHO, 2005).

3.2 UTILIZANDO ÍNDICES EFETIVAMENTE

Na seção acima, conforme destacado por Milani (2008), foi mencionada a importância

da indexação no banco de dados. No entanto, além de criar os índices é importante garantir que as consultas sejam projetadas adequadamente para que utilizem os índices de forma eficaz (FRITCHEY & DAM, 2009).

O SQL *Server* classifica seus predicados de consulta em dois tipos: *sargable* e *nonsargable*. A palavra *sargable* é originada de "*search argument able*", que significa que o predicado que está sendo utilizado com a cláusula *where* pode fazer uso dos índices, caso as colunas requeridas o tiverem. A tabela 1 mostra os respectivos predicados de cada tipo.

Tabela 1: Predicados de pesquisa *sargable* e *nonsargable*
Fonte: Fritchey & Dam (2009)

Tipo	**Significado**	**Condição de Pesquisa (predicados)**
Sargable	Faz uso de índices	Inclui os predicados: =, >, >=, <, <=, *between* e a condição *like* no formato '<literal>%'
Nonsargable	Não faz uso de índices	Exclui os predicados: <>, !=, !>, !<, *is null*, *not exists*, *not in*, *not like in*, *in*, *or,* a condição *like* no formato '%<literal>' e uso de funções internas

O SQL *Server*, conforme destaca Pilecki (2007), "utiliza um otimizador de consultas baseado em custo, ou seja, ele tenta gerar um plano de execução com o menor custo estimado. Essa estimativa se baseia na estatística de distribuição dos dados disponíveis para o otimizador quando ele avalia cada tabela envolvida na consulta". Com isso, o otimizador pode ser habilitado a fazer uso de um índice dependendo dos predicados utilizados na seletividade dos dados na consulta, criando assim, um plano de consulta de alta qualidade (FRITCHEY & DAM, 2009; MSDN, 2009).

Nas próximas subseções serão exemplificadas e comparadas algumas situações em que o SQL *Server* não considera o uso do índice devido aos predicados da cláusula *where*. As situações são[2]: uso da condição *like*, *in*/*or* e uso de funções internas. Para isso será utilizado o recurso de visualização do plano de execução das consultas que, conforme Pilecki (2007) "descreve a sequência de operações, físicas e lógicas, que o SQL *Server* executa para fazer a consulta e gera o conjunto de resultados desejado".

2 A escolha por esses predicados deve-se ao fato de serem os mais utilizados na empresa, que disponibilizou o banco de dados para realização de *tuning*.

3.2.1 USANDO O OPERADOR *LIKE*

O operador de comparação *like* cria condições para comparações de partes de uma cadeia de caracteres. Estas podem ser especificadas utilizando o caractere reservado '%', que substitui um número arbitrário entre zero ou mais caracteres (ELMASRI & NAVATHE, 2005). Como exemplo pode-se citar as consultas abaixo, que retornam todos os produtos cujo nome cadastrado inicie com a expressão “Hex Nut” ou que finalizem com a expressão “Nut 3”, respectivamente representadas pelos planos de execução demonstrados na Figura 5A e Figura 5B.

```
A) SELECT ProductID,Name
        FROM Production.Product   WHERE Name like 'Hex Nut%'

B) SELECT ProductID,Name
        FROM Production.Product WHERE Name like '% Nut 3'
```

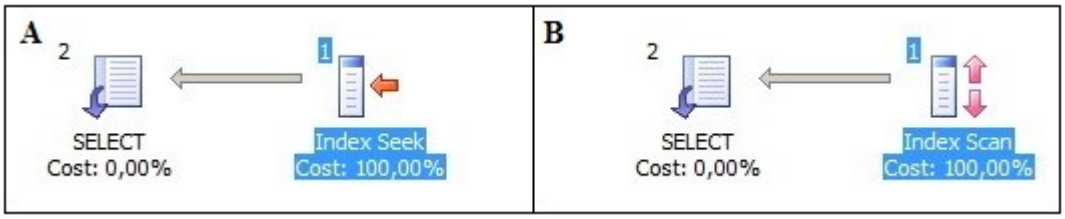


Figura 5: Plano de execução utilizando o predicado *like*
Fonte: Autor

Quando o predicado *like* é utilizando no formato <literal>%, o otimizador de consulta consegue utilizar o índice da coluna específica para agilizar a consulta. O plano de execução da SQL A, confirma que o índice utilizado foi do tipo *index seek,* que, segundo Ribeiro(2004), “realiza uma pesquisa, que por sua vez é pontual e específica, em um índice não-clusterizado, lendo somente as informações necessárias, o que deixa a consulta muito mais eficiente”. Segundo Fritchey & Dam (2009) “quanto maior o número de caracteres na condição *like*, melhor o otimizador irá trabalhar em busca de um índice eficaz”.

Porém, se esse mesmo predicado for utilizado no formato %<literal>, o otimizador não irá utilizar o índice da coluna *Name*, pois ele só pode encontrar o índice apropriado se os caracteres iniciais forem informados. Com isso, será executada uma varredura completa na tabela para retornar as informações desejadas, conforme pode ser visualizado na Figura 5B,

onde mostra o plano de execução da consulta, identificando que o índice utilizado foi do tipo *index scan*. De acordo com Ribeiro (2004), esse tipo de índice "realiza uma busca, em forma sequencial, em todos os elementos de um índice não-clusterizado, sendo assim esse tipo de busca deve ser evitado o máximo possível".

3.2.2 USANDO O OPERADOR *IN*/*OR*

O operador *in* determina se um valor especificado corresponde a qualquer valor em uma subconsulta ou uma lista. Enquanto que o operador *or* é a combinação de duas condições lógicas (MSDN, 2009). Apesar de os operadores não terem uma definição semelhante, quando uma consulta é executada utilizando o operador *in*, o otimizador o transforma em operações do tipo *or*. Sendo assim, se a condição *in* possui dez valores associados a ela, o SQL *Server* os transforma em dez condições *or* (FRITCHEY & DAM, 2009).

Para melhorar a performance, pode-se trocar os operadores *in*/*or* por *between*, que também é um operador lógico que especifica um intervalo de resultados a serem buscados (FRITCHEY & DAM, 2009; MSDN, 2009).

Para ilustrar essa situação, os comandos SQL abaixo serão utilizados como exemplos. As consultas retornam o número do pedido, o código do produto e seu subtotal, referentes aos pedidos com identificador entre 51800 e 51829. Algumas considerações importantes devem ser feitas sobre os campos utilizados: o atributo *SalesOrderId* possui um índice clusterizado; o atributo *ProductiID* possui um índice não-clusterizado e, o atributo *LineTotal* é um valor computado que representa o subtotal do produto, e não possui índice associado a ele.

```
A) SELECT SalesOrderID, ProductID, LineTotal
                FROM Sales.SalesOrderDetail
        WHERE SalesOrderID IN
                (51800,51801,51802,51803,51804,51805,51806,51807,51808,51809,
                51810,51811,51812,51813,51814,51815,51816,51817,51818,51819,
                51820,51821,51822,51823,51824,51825,51826,51827,51828,51829)

B) SELECT SalesOrderID, ProductID, LineTotal
                FROM Sales.SalesOrderDetail
        WHERE SalesOrderID BETWEEN 51800 and 51829
```

A Figura 6 mostra os planos de execução gerados para as consultas utilizando o predicado *in* (A) e o predicado *between* (B).

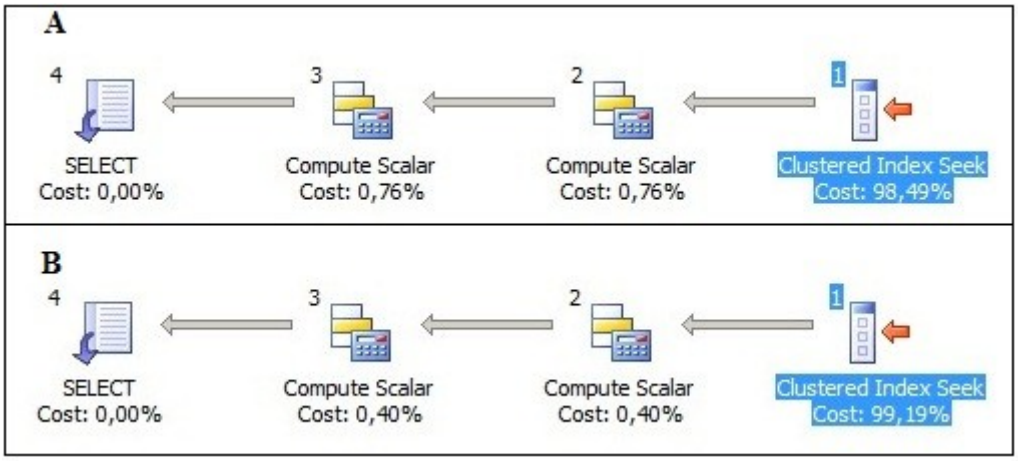


Figura 6: Plano de execução utilizando os predicados *in* e *between*
Fonte: Autor

Como pode-se visualizar, o plano de execução é o mesmo, passando pelos mesmos caminhos até retornar os dados desejados. Porém, o custo associado ao atributo *LineTotal*, indicado nos passos 2 e 3, para calcular o valor do campo em cada linha pesquisada, é a diferença entre os predicados *in*, onde o custo é de 0,76% , e *between*, onde o custo diminui para 0,40%. Isso se deve ao fato do número estimado de linhas que foi produzido por cada predicado: o operador *in* produziu 587 linhas, enquanto que o *between* produziu 302. Sendo assim, quanto mais linhas a ser pesquisados, maior é custo atribuído a essa operação.

3.2.3 UTILIZANDO FUNÇÕES

No SQL *Server*, assim como outros SGBDs, é possível fazer uso de funções internas, que também podem ser criadas pelos usuários (MSDN, 2009). As funções podem facilitar o cotidiano dos desenvolvedores, porém seu uso na estrutura do comando SQL interfere no uso de índices no campo que esta recebendo a função.

O plano de execução gerado com o comando SQL, conforme a Figura 7A, ilustra a afirmação acima. As consultas A e B retornam o nome e o *e-mail* dos clientes que possuem o primeiro nome “Sabrine”, sendo que a primeira consulta utiliza a função interna *upper*.. Ambos os campos retornados possuem índice não-clusterizado.

A) SELECT FirstName, EmailAddress
FROM Person.Contact WHERE UPPER(FirstName)='SABRINE'

B) SELECT FirstName, EmailAddress
FROM Person.Contact WHERE FirstName='Sabrine'

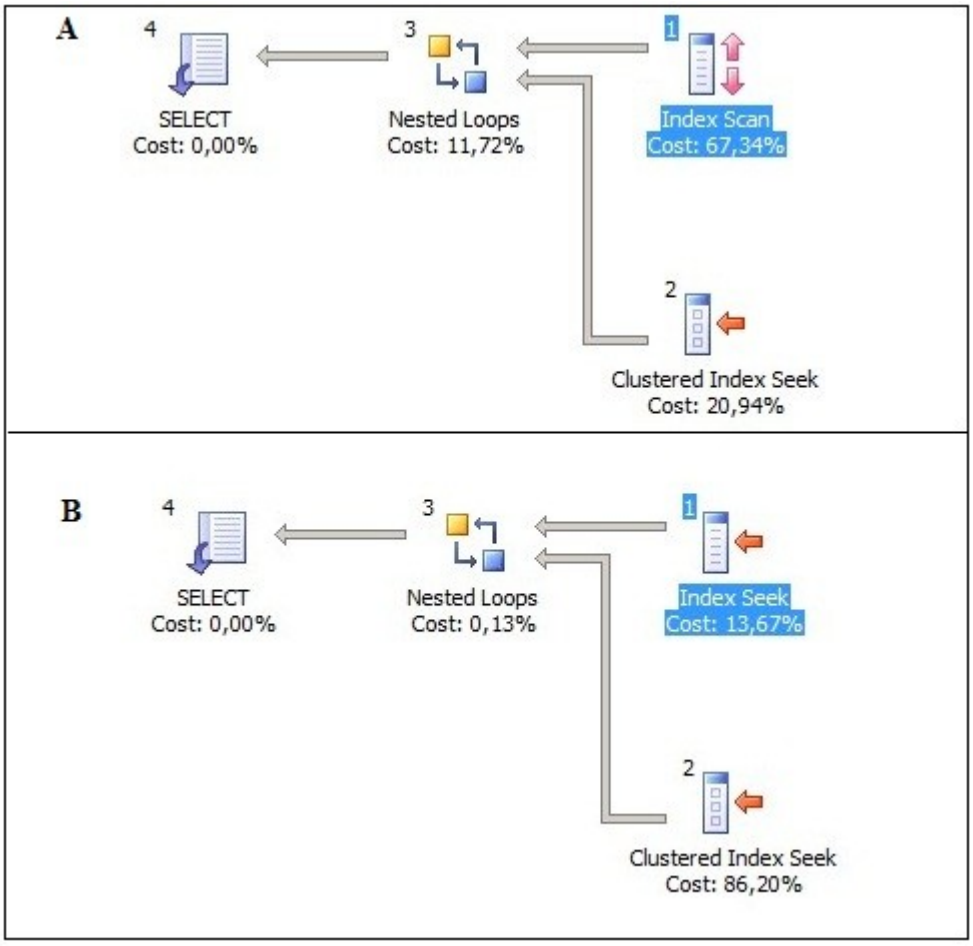


Figura 7: Plano de execução utilizando a função interna *upper*
Fonte: Autor

O primeiro passo, do plano de execução, é a pesquisa pelo índice no campo *FirstName*, e como pode ser visualizado, na Figura 7A, o índice utilizando foi do tipo *scan*, quando utilizada a função *upper* relacionada com o campo. No entanto, quando a função é retirada, o tipo de índice que passa a ser utilizado é do tipo *seek*, que possui um menor custo, conforme representado na Figura 7B.

Na próxima seção será descrito o uso de junções de modo a otimizar as consultas.

3.3 JUNÇÕES

O uso de junções permite, segundo o MSDN (2009), "recuperar dados de duas ou mais tabelas com base em relações lógicas entre elas". Na grande maioria dos casos, utilizar uma só consulta geralmente é mais eficiente do que duas (PRICE, 2009). Por isso, deve-se fazer o uso de junções sempre que se deseja retornar dados de tabelas distintas, tendo o cuidado para que a junção realizada não prejudique o desempenho da consulta.

Nas próximas subseções serão detalhadas as estratégias que o otimizador de consulta do SQL *Server* utiliza para realizar as junções, sendo elas: *hash join*, *merge join* e *nested loop join*. Também será conceitualizado os três tipos de junções: internas (*inner*), externas (*outer*) e cruzadas (*cross*), que serão utilizados nas consultas de exemplo. Além disso, serão citadas algumas considerações importantes para que o uso de junções se torne eficiente.

3.3.1 *HASH JOIN*

A junção de *hash* é executada em duas fases conhecidas por contrução e execução. Na fase de construção, o otimizador escolhe a tabela com menor número de tuplas para construir uma tabela em memória, tabela *hash*. Para cada dado inserido na tabela *hash*, é aplicada a função *hash*, que, de acordo com o MSDN(2009), "dado um parâmetro de entrada será retornado um valor de saída de forma determinística, ou seja, um valor produzirá sempre o mesmo *hash*". Após, na fase de execução, será lida cada linha da tabela que não está em memória, gerando o valor *hash* e verificado se o valor *hash* gerado tem o mesmo valor da tabela *hash*, retornando assim todas os registros que possuem o mesmo valor. O relacionamento entre as tabelas é realizado pela chave de junção, que são os campos que compõe o relacionamento entre as tabelas (FRITCHEY & DAM, 2009). Esses passos podem ser visualizados na Figura 8, que mostra o plano de execução gerado utilizando *hash join*.

A escolha pelo tipo de armazenamento que será utilizado é baseado nas estatísticas geradas pelo otimizador, por isso nem sempre o SQL *Server* irá armazenar a tabela *hash* em memória, podendo assim, ser combinadas técnicas de armazenamento em disco (MSDN, 2009).

Para exemplificar o uso de junções do tipo *hash* será utilizada uma junção interna. A junção interna é uma junção na qual os valores associados das colunas são comparados,

utilizando um operador de comparação para relacionar duas tabelas, com base nos valores em colunas comuns a cada uma (MSDN, 2009). Sendo assim, apenas as linhas da tabela que satisfaça as condições de junção são usadas para construir o conjunto de resultados (SACK, 2005). Esse tipo de junção pode ser especificada tanto na cláusula *from* quanto na *where*, e utiliza a palavras-chaves *inner join* para criar a junção. Porém, pode ser utilizado somente a palavra-chave *join*, que junção interna fica subentendida (MSDN, 2009).

A seguir, a consulta que será utilizada como exemplo, bem como seu plano de execução gerado.

```
SELECT p.ProductID, p.ProductSubcategoryID, p.Name
    FROM Production.Product p
    INNER JOIN Production.ProductSubcategory s
ON p.ProductSubcategoryID= s.ProductSubcategoryID
```

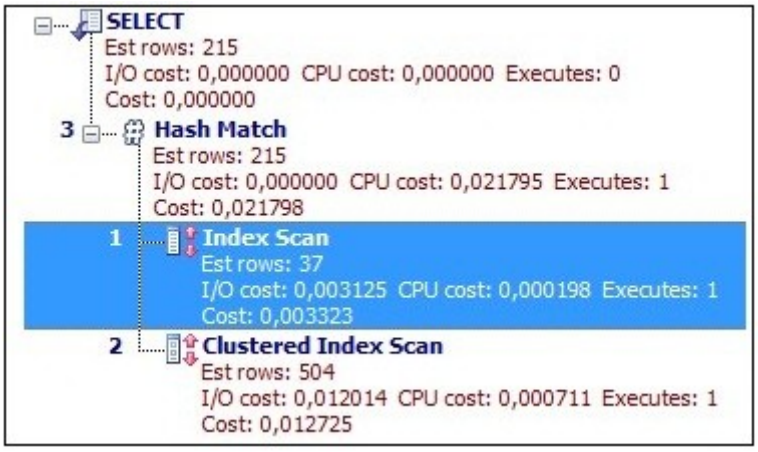


Figura 8: Plano de execução utilizando *hash join*
Fonte: Autor

A estratégia de *hash join* será utilizada no pior caso, quando não há índices suficientes nas colunas que estão sendo utilizadas para junção entre as tabelas. É o caso da consulta utilizada como exemplo, onde dos campos de junção entre as tabelas, apenas o campo *ProductSubcategoryID* da tabela *ProductSubcategory* possui índice.

3.3.2 *MERGE JOIN*

A junção do tipo *merge*, primeiramente, ordena todas as linhas relevantes de cada tabela, de acordo com o campo chave que esta sendo utilizado na junção. Após, lê e compara,

simultaneamente, cada linha da tabela mais interna com a tabela mais externa, cujo atributo da relação de junção coincide. As linhas que corresponderem aos critérios de junção, serão adicionadas ao conjunto resultado. Esse processo é realizado até que todas as linhas sejam processadas (FRITCHEY & DAM, 2009).

Para o tipo de junção *merge* ser um plano de consulta realmente eficiente, no SQL *Server*, algumas premissas devem ser levadas em consideração. A primeira delas refere-se ao tamanho das tabelas ser semelhante. E a segunda refere-se as tabelas possuirem índices clusterizados com base nas colunas de junção, assim não há necessidade de ordenar os dados previamente (MSDN, 2009).

Como exemplo, será utilizado o comando SQL abaixo, que retorna todos os funcionários cadastrados, mesmo aqueles que não possuem endereço relacionado. Sendo assim, na consulta foi utilizado uma junção externa que, de acordo com o MSDN (2009) "retorna linhas apenas quando há pelo menos uma linha nas tabelas que corresponde à condição de junção". A principal diferença entre junções internas e externas é que a interna elimina as linhas que não correspondem a uma linha da outra tabela, enquanto que na junção externa é retornado todas as linhas de pelo menos uma das tabelas menciondadas na cláusula *from*, contanto que essas linhas atendam a algum critério de pesquisa *where* (MSDN, 2009).

A Figura 9 mostra o plano de execução gerado com a consulta, sendo que para realizar a junção entre as tabelas foi utilizado o *merge join*.

```
SELECT e.EmployeeID,a.AddressID
        FROM HumanResources.Employee e
        LEFT OUTER JOIN HumanResources.EmployeeAddress a
ON e.EmployeeID = a.EmployeeID
```

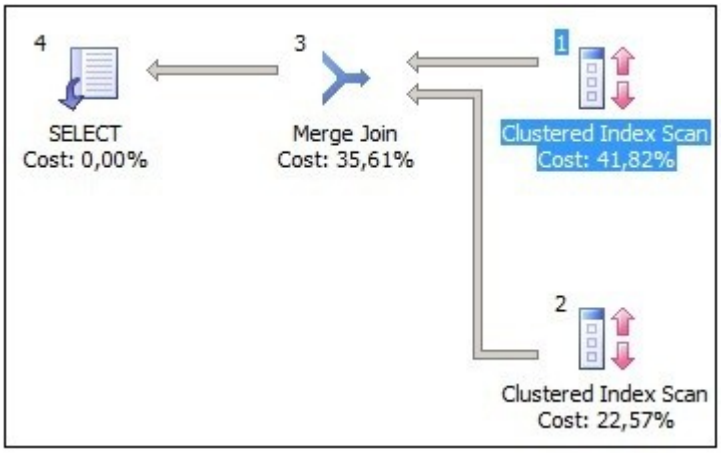


Figura 9: Plano de execução utilizando *merge join*
Fonte: Autor

O campo *EmployeeID*, utilizado como campo de junção, possui um índice clusterizado em ambas as tabelas, sendo que na tabela *Employee* é o campo que identifica a chave primária (PK) e, na tabela *EmployeeAddress*, é uma chave estrangeira (FK).

3.3.3 *NESTED LOOP JOIN*

O tipo de junção *nested loop* é, de acordo com Fritchey & Dam (2009) "altamente eficaz se a entrada exterior é muito pequena e a entrada interna é grande mas indexada. Em muitas consultas simples, que afetam um pequeno conjunto de linhas, a junção por *nested loop* é muito superior às junções do tipo *hash* e *merge*".

Quando o otimizador decide utilizar esse tipo de junção, é feita uma divisão entre os dois operandos da operação de junção (tabelas) em tabela interna e tabela externa. Para cada linha da tabela interior, é percorrida a tabela exterior e feita a comparação segundo as condições de junção definidas. Assim, as linhas que satisfaçam as condições são selecionadas (MSDN, 2009).

O comando SQL, utilizado como exemplo, retorna todos os representantes relacionados com todas os territórios de vendas. Por isso, o tipo de junção utilizado foi o *cross join*, que retorna um produto cartesiano que, por sua vez, produz um resultado definido com base em todas as combinações possíveis de linhas da tabela à esquerda, multiplicando contra as linhas na tabela à direita (SACK, 2005).

É possível visualizar na Figura 10, que ilustra o plano de execução da consulta, que foram gerados 170 registros, sendo esses resultado da multiplicação dos 17 registros da tabela *SalesPerson* com os 10 registros da tabela *SalesTerritory*. Para junções cruzadas, não é necessário possuir uma coluna em comum para realizar a junção entre as tabelas. Porém, se for utilizada a cláusula *where*, conforme o MSDN(2009) “a junção cruzada se comportará como uma junção interna”. Esse tipo de junção não possui um bom desempenho, devido ao produto cartesiano que precisa ser realizado, por isso, seu uso é recomendado somente em casos bem específicos.

```
SELECT p.SalesPersonID, t.Name as Territory
        FROM Sales.SalesPerson p
        CROSS JOIN Sales.SalesTerritory t
ORDER BY p.SalesPersonID
```

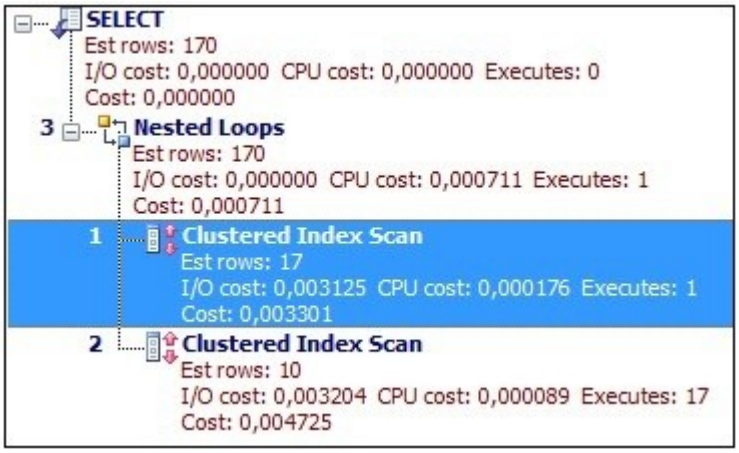


Figura 10: Plano de execução utilizando *nested loop join*
Fonte: Autor

Conforme pode ser visualizado na Figura 10, foi utilizado a junção do tipo *nested loop* para unir o resultado das duas tabelas envolvidas na consulta. Mesmo não tendo especificado uma coluna de junção entre as tabelas definidas na consulta, para pesquisar os dados em cada tabela foi utilizado a estrutura de *clustetered index scan*, pois ambas possuem um índice clusterizado.

3.3.4 CONSIDERAÇÕES SOBRE JUNÇÕES

Com a utilização de junções, os comandos SQL passam a não ser mais triviais, principalmente, quando o número de tabelas envolvidos passa de duas. Sendo assim, a utilização de algumas boas práticas é importante, podendo ser destacado algumas que são de fácil implementação:

- Incluir em todas as colunas da consulta a tabela a que se refere, assim o banco de dados não precisa procurar nas tabelas cada coluna que foi utilizada na consulta. Esse tempo gasto na pesquisa pela tabela, é tempo desperdiçado (PRICE, 2009). Para auxiliar nesse item, pode-se fazer o uso de *alias*, que nada mais são que apelidos para as tabelas, facilitando assim a escrita da consulta, pois não será preciso informar o nome completo da tabela, apenas seu apelido;

- Procurar restringuir o número de colunas que devem ser retornadas, evitando o uso de

asterisco (*), além de fazer uso da cláusula *where*, restringindo o número de linhas que devem ser retornadas. Quanto mais linhas estiverem envolvidas na junção, mais comparações devem ser realizadas (MCGEHEE, 2006a; PRICE, 2009);

- Antes de escolher o tipo de *join* (*inner*, *outer* ou *cross*) para ser utilizado na consulta, deve-se saber que resultados são esperados. O uso de um tipo incorreto, além de não retornar os dados necessários, pode influenciar na performance da consulta (MCGEHEE, 2006a);

- Criar índices, se possível exclusivos, nos campos utilizados nas junções entre as tabelas melhora muito o desempenho da consulta (MCGEHEE, 2006a; PRICE, 2009);

- A utilização de subconsultas, geralmente, é mais eficiente somente quando a quantidade de dados retornados é pequena ou se não houver índices nas colunas de junção. Caso contrário, a recomendação é utilizar um *join* (MCGEHEE, 2006a). Porém, a única maneira de saber realmente qual é a melhor forma é executando os dois modos e analisando o plano de execução gerado (MILANI, 2008)

A análise do plano de execução gerado, auxilia na identificação de possíveis gargalos durante a execução da pesquisa. Porém, é importante ressaltar, que o desempenho das consultas não depende somente da forma como os comandos SQL são escritos, outros fatores, com *hardware*, rede e estrutura das tabelas, influenciam diretamente na performance da aplicação.

3.4 ORDENAÇÃO E AGRUPAMENTO

A cláusula *order by* é utilizada para ordenar o resultado da consulta em ordem crescente ou decrescente, utilizando o parâmetros *asc* ou *desc*, respectivamente. Porém, esse tipo de operação pode ter um alto custo. O desempenho das ordenações são afetas pelos seguintes fatores, por ordem de importância: número de linhas selecionadas, número de colunas e tamanho das colunas na cláusula *order by* (GULUTZAN & PELZER, 2003). Se a cláusula não for especificada na consulta, a ordenação é determinada pela ordem física da

coluna chave do índice clusterizado (SACK, 2005).

Internamente, o SQL *Server* utiliza os algoritmos de ordenação quando realiza operações para criar índices exclusivos, agrupamento ou agregação de dados através dos comando *group by* e *distinct* e, junção de tabelas utilizando o método *merge join*. Sendo assim, o recomendado é não utilizar a cláusula *order by* se a consulta possui um *distinct* ou *group by*, pois a operação torna-se redundante (GULUTZAN & PELZER, 2003). Além disso, também recomenda-se evitar o uso de muitos atributos na cláusula *order by*, pois a cada atributo inserido nessa cláusula é realizada uma ordenação interna (SACK, 2005).

Já para realizar o agrupamento dos dados, é utilizado o operador *group by*, que agrupa linhas com valores comuns e retorna um resumo das linhas para cada grupo (MSDN, 2009). Para otimizar o uso dessa cláusula, recomenda-se criar um índice com os atributos que fazem parte do *group by* e os atributos agregados (FRITCHEY & DAM, 2009). Na consulta abaixo, por exemplo, deve-se criar um índice composto pelos campos *LineTotal* e *PurchaseOrderID*.

```
SELECT sum(LineTotal),PurchaseOrderID
        FROM Purchasing.PurchaseOrderDetail
GROUP BY PurchaseOrderID
```

Juntamente com o *group by* pode ser utilizado o *having*, que especifica um critério de pesquisa para um grupo ou um valor agregado (MSDN, 2009). Apesar de ter quase a mesma funcionalidade do *where*, o *having* elimina linhas depois da agregação, enquanto que o *where* elimina essas mesmas linhas antes que o dados sejam agrupados (SACK, 2005). Por isso, o uso do *where*, quando possível, deve ser priorizado, tendo assim um menor número de linhas participando de junções e ordenação, melhorando o desempenho da consulta.

Ao realizar ordenações e agrupamentos, deve-se sempre pesar os prós e contras dessas operações, para evitar desperdícios de recursos e tempo, melhorando assim, o desempenho das consultas.

3.5 *HINTS*

Os *hints* de acordo com o MSDN (2009), são "estratégias especificadas para aplicação pelo processador de consultas do SQL *Server* em instruções *select*, *insert*, *update* ou *delete*.

Substituem todos os planos de execução que o otimizador de consulta possa selecionar para uma consulta". Mas deve-se tomar cuidado e ser bastante criterioso ao fazer a escolha pela utilização de *hints*, pois o otimizador de consulta seleciona, geralmente, o melhor plano de execução. A escolha errada de um *hint* irá prejudicar o desempenho do comando SQL em vez de colaborar com um resultado mais rápido (MSDN, 2009; MCGEHEE, 2006b).

O uso de *hints* pode ser justificado em um determinado conjunto de circunstâncias. Mas se essas circunstâncias mudarem, os *hints* podem não ser mais apropriados. Por isso, deve-se estabelecer um processo de revisão dos *hints* periodicamente, para que seu objetivo inicial seja realmente alcançado (MCGEHEE, 2006b; SACK, 2005).

Nas próximas seções serão detalhadas os tipos de *hints* disponíveis no SQL *Server*, sendo eles: *hints* de junções, *hints* de tabelas e *hints* de consulta. Serão destacados apenas alguns *hints* disponíveis, sendo que a lista completa pode ser encontrada no endereço eletrônico http://msdn.microsoft.com/en-us/library/ms187713(v=SQL.105).aspx.

3.5.1 *HINTS* DE JUNÇÃO

Os *hints* de junção especificam que o otimizador de consulta force o uso de uma estratégia de junção entre as tabelas (MSDN, 2009). Sendo assim, esse tipo de *hint* indica como os dados deverão ser unidos, restando ao otimizador escolher o melhor caminho para retornar os dados da consulta. Atualmente, no SQL *Server*, estão disponíveis os tipos de *hints* para junção apresentados na tabela 2.

Tabela 2: *Hints* de junção disponíveis no SQL *Server*
Fonte: MSDN (2009); Sack (2005)

Hint	**Tipo de junção equivalente**	**Utilização Ideal para o *Hint***
Loop join	*Nested loop*	Quando uma tabela é pequena e a outra é grande, porém existe um índice nas colunas que unem as tabelas.
Hash join	*Hash*	Quando ambas as tabelas são grandes, porém não existe índice nas colunas utilizadas na junção.
Merge join	*Merge*	Quando as tabelas são de médio ou grande porte e a ordenação é realizada através da coluna de junção. Útil também quando a quantidade de linhas retornas é grande.

A seguir uma consulta que originalmente seria executada utilizando *nested loop*, com o uso de *hint* passará a ser executada utilizando *hash join*.

```
SELECT p.Name,r.ReviewerName,r.Rating
        FROM Production.Product p
        INNER JOIN Production.ProductReview r
    ON r.ProductID= p.ProductID
    OPTION (HASH JOIN)
```

A inserção do *hint* em uma junção pode ser realizada de duas maneiras. A primeira delas é inserir a cláusula *option* (*loop join* | *merge join* | *hash join*) ao fim do comando SQL, conforme o exemplo acima. Utilizando dessa maneira a dica se expandirá a todas as junções especificadas, ou seja, se a consulta possui duas junções, ambas serão executadas com o *hint* de junção definido. O segundo modo é inserir o *hint* em cada junção, conforme a consulta abaixo, assim cada uma pode ser executada com um *hint* específico.

```
SELECT p.Name,r.ReviewerName,r.Rating
        FROM Production.Product p
        INNER HASH JOIN Production.ProductReview r
    ON r.ProductID= p.ProductID
```

3.5.2 *HINTS* DE TABELAS

Os *hints* de tabelas, no SQL *Server*, de acordo com o MSDN (2009), "substituem o comportamento padrão do otimizador de consulta durante a instrução DML (Linguagem de Manipulação de Dados) ao especificar, por exemplo, um método de bloqueio, um ou mais índices ou uma operação de processamento de consulta".

Porém, nem sempre o otimizador de consulta utiliza os *hints* de tabelas especificados. Eles podem ser ignorados caso a tabela, que possui a dica, não for acessada pelo otimizador ou porque uma exibição indexada[3] foi acessada (MSDN, 2009).

A seguir serão destacados três *hints*, disponíveis no SQL *Server*, que auxiliam no uso

3 É uma exibição (*view*) com índice clusterizado exclusivo. Se uma consulta tiver referências a colunas presentes em uma exibição indexada e em tabelas bases, e o otimizador de consulta determinar que o uso da exibição indexada oferece o melhor método para execução da consulta, o otimizador poderá utilizar o índice na exibição. Porém, essa funcionalidade só é suportada nas edições *Enterprise* e *Developer*, do SQL *Server* (MSDN, 2009).

de índices, processamento de consultas e método de bloqueio/isolamento de tabelas, respectivamente:

- *Index*: De modo geral, o otimizador do SQL *Server*, escolhe corretamente os índices a serem utilizados nas consultas. Mas, para os casos em que um índice venha a ser ignorado, pode-se utilizar este *hint*, que tem como objetivo forçar que o otimizador de consulta utilize um ou mais índices. Apenas uma dica de índice pode ser especificada por tabela. Porém, cada dica pode conter vários índices, sendo que os índices com maior cobertura devem ser informados primeiros. Se a lista de índices informados no *hint* não incluir todas as colunas referidas pela consulta, será executada um busca para recuperar as colunas restantes, depois que o SQL *Server* recuperar todas as colunas indexadas (MSDN, 2009). A sintaxe para utilização desse *hint* consta no comando SQL abaixo:

```
SELECT PurchaseOrderID,PurchaseOrderDetailID,ProductID,LineTotal
        FROM Purchasing.PurchaseOrderDetail
WITH(INDEX (PK_PurchaseOrderDetail_PurchaseOrderID_PurchaseOrderDetailID))
        WHERE LineTotal >100
```

- *ForceSeek*: Esse *hint* força o otimizador de consulta a usar só uma operação de busca de índice como caminho de acesso para os dados na tabela referenciada na consulta. O *forceseek* aplica-se as operações de busca de índice clusterizado e não-clusterizado. Pode ser especificado apenas uma vez, em qualquer tabela, na cláusula *from* de uma instrução *select, update* ou *delete*, sendo que também pode ser utilizado com o *hint index*, citado acima, assim, além de forçar a utilização de um índice especifico, irá forçar uma operação de busca sobre eles (MSDN, 2009; FRITCHEY & DAM, 2009). A seguir, a sintaxe de utilização desse *hint*, exemplificada na consulta:

```
SELECT c.CustomerID
        FROM Sales.Customer c WITH (FORCESEEK)
        INNER JOIN Sales.CustomerAddress ca
    ON c.CustomerID=ca.CustomerID
 WHERE c.TerritoryID <> 1
```

- *ReadCommitted*: Pertence a classificação, de *hints* de tabelas, que determina o nível de isolamento da tabela (SACK, 2005). Especifica que as operações de leitura obedecem a regras de nível de isolamento *readcommitted* usando bloqueio ou controle de versão de linha, conforme configurado no SQL *Server*. Esse nível de isolamento é o padrão do SQL *Server*, que quando utilizado usará bloqueios compartilhados durante a leitura de dados, ou seja, ele garante que um dado que ainda não foi gravado fisicamente no banco de dados, não vá ser lido (MSDN, 2009). A seguir, a sintaxe utilizada para esse *hint*, sendo que ela pode ser definida em cada tabela especificada no *select*:

```
SELECT sum(TotalDue)
        FROM  Sales.SalesOrderHeader WITH (READCOMMITTED)
WHERE CreditCardID=4503
```

Já o *hint nolock* faz exatamente o contrário, permite que dados que ainda não foram comitados no servidor sejam lidos nas consultas. As dicas de bloqueio são propagadas para todas as tabelas acessadas pelo plano de consulta, sendo que o SQL *Server* executa os teste de consistência de bloqueio correspondentes (MSDN, 2009). A seguir a sintaxe para utilização desse *hint*:

```
SELECT sum(TotalDue)
        FROM  Sales.SalesOrderHeader  WITH (NOLOCK)
WHERE CreditCardID=4503
```

3.5.3 *HINTS* DE CONSULTA

Os *hints* de consulta, de acordo com (SACK, 2005), "são instruções enviadas, junto com as consultas, ao SQL *Server*, para substituir uma decisão do otimizador de consulta". Os *hints* afetam todos os operadores na instrução, podendo ser utilizado em *select*, *update* e *delete*. Na operação de *insert*, só poderá ser utilizada se um *select* fizer parte da instrução. Os *hints* são válidos apenas para consultas de nível superior, ou seja, as subconsultas não são afetadas pelos *hints* (MSDN, 2009).

A seguir serão destacados três *hints* de consultas, disponibilizados pelo SQL *Server* 2008:

- *Fast*: Especifica que a consulta é otimizada para recuperação rápida das *n* primeiras linhas, assim é possível trabalhar com as *n* primeiras linhas retornadas até que se produza o resultado completo da instrução *select* (MSDN, 2009; SACK, 2005). A consulta a seguir demonstra a sintaxe para utilização desse *hint*.

```
SELECT Name
        FROM Production.Product
OPTION (FAST 100)
```

- *Hash | Order Group:* Esse *hint* especifica que as agregações, especificadas na cláusula *group by* e *distinct*, devem usar *hash* ou ordenação (MSDN, 2009). Assim como os demais *hints*, a sintaxe de utilização desse também é simples, como pode ser visualizada na consulta abaixo.

```
SELECT ProductID, OrderQty, sum(LineTotal) as Total
        FROM Sales.SalesOrderDetail
WHERE UnitPrice < 5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP)
```

- *Recompile*: Esse *hint* força que o SQL *Server* descarte o plano gerado para a consulta depois de sua execução, sendo assim, a próxima vez que a consulta for executada, o otimizador será forçado a recompilar um novo plano de consulta (SACK, 2005). Se o *recompile* não for especificado, de acordo com o MSDN (2009), o SQL *Server* "armazena em *cache* os planos de consulta e os reutiliza".

  O SQL *Server* disponibiliza a opção *with recompile*, que não se trata de um *hint*, utilizada em procedimentos armazenados indicando que o mesmo deve ser recompilado a cada execução. Essa opção é utilizada principalmente em procedimentos armazenados, pois eles possuem uma grande variação de parâmetros de entrada, assim, realizando a recompilação a cada execução, o plano de consulta não é afetado. Porém uma desvantagem de utilizar a opção *with recompile* é que afeta todas as consultas envolvidas no procedimento, o que pode acarretar em uma carga extra de recursos do SQL *Server*. Para isso, com a versão lançada em 2005 do SQL *Server*, foi criado o *hint recompile*, onde é possível especificar que apenas algumas consultas devam ser recompiladas, poupando tempo de recompilação e recursos (MSDN, 2009;

MCGEHEE, 2007). Sendo assim, o exemplo a seguir demonstra a sintaxe de utilização para esse *hint*:

```
CREATE PROCEDURE dbo.uspProductByVendor @Name varchar(30) = '%'  AS
    SELECT v.Name AS 'Vendor name', p.Name AS 'Product name'
            FROM Purchasing.Vendor as v
    JOIN Purchasing.ProductVendor as pv   ON v.VendorID = pv.VendorID
    JOIN Production.Product AS p   ON pv.ProductID = p.ProductID
    WHERE v.Name LIKE @Name
    OPTION (RECOMPILE)
```

3.6 CONSIDERAÇÕES FINAIS

Inicialmente, o capítulo apresentou os tipos de índices disponíveis no SQL *Server*, especificando também seu uso de forma eficiente, através da aplicação de alguns exemplos. Após, foram abordadas as três estratégias de junções, utilizadas pelo otimizador de consulta do SQL *Server*, utilizando os planos de consulta gerados para exemplificar a aplicação de cada estratégia. Seguindo, o capítulo apresentou uma abordagem sobre a utilização de operações de ordenação e agrupamento de forma a otimizar as consultas realizadas. Por fim, foi apresentado uma pequena amostra dos *hints* disponibilizados pelo SQL *Server*, que contribuem para um melhor desempenho dos comandos SQL. A tabela 3 mostra um resumo desses assuntos abordados.

Tabela 3: Resumo dos assuntos abordados no capítulo
Fonte: Autor

Assunto	**Característica**	**Quando é utilizado**
Índice clusterizado	Organiza as linhas das tabelas na mesma ordem das entradas do índice. Deve ser utilizado em atributos que não possuam muitas operações de atualização	No índice referente a chave primária da tabela
Índice não-clusterizado	Tem a estrutura separada da tabela, onde a ordem física das linhas não segue a ordem do arquivo de índice	Nos demais índices criados na tabela
Índice exclusivo	Garante que a chave do índice não contenha valores duplicados	Quando necessita-se que se tenha exclusividade sobre os atributos que estão sendo indexados

Assunto	Característica	Quando é utilizado
Predicado *Sargable*	Predicados desse tipo podem fazer uso de índices, caso o atributo requerido o tenha	Predicados com essa característica são definidos na Tabela 1
Predicado *Nonsargable*	Predicados desse tipo não fazem uso de índices, caso o atributo requerido o tenha	Predicados com essa característica são definidos na Tabela 1
Hash join	Executada em duas fases: construção, onde o otimizador escolhe a tabela com menor número de tuplas para construir uma tabela em memória, sendo que para cada dado inserido é executada a função *hash;* e execução, onde será lido cada linha da tabela que não esta na memória, gerando o valor *hash* e verificado se o valor *hash* gerado tem o mesmo valor da tabela *hash*, retornando somente os registros que possuem o mesmo valor	Quando não há índices suficientes nas colunas que estão sendo utilizadas para a junção
Merge join	Primeiramente ordena todas as linhas relevantes de cada tabela, de acordo com o campo chave utilizando na junção. Após, lê e compara, simultaneamente, cada linha da tabela mais interna com a tabela mais externa, cujo atributo da relação de junção coincide	As tabelas envolvidas na junção, devem possuir índice clusterizado, com base nos atributos de junção
Nested loop join	Realizada uma divisão entre os dois operadores da junção em tabela interna e externa. Para cada linha da tabela interna, é percorrida a tabela externa e feita a comparação segundo as condições de junção definidas, sendo que as linhas que satisfaçam a condição são selecionadas	Altamente eficaz se a entrada exterior é muito pequena e a entrada interna é grande mas indexada

Boas práticas de estruturação do comando SQL foram mencionadas neste capítulo. Algumas delas serão destacadas a seguir:

- Procurar utilizar, sempre que possível, predicados classificados como *sargable*, assim o otimizador de consultas pode fazer uso dos índices dos atributos, caso esses o tenham, utilizados na cláusula *where*;

- Restringir o número de colunas que devem ser retornadas, incluindo em todas as colunas o nome ou apelido da tabela referida;

- Para realizar junção, utilizar atributos do tipo numérico;

- Não recomenda-se utilizar a cláusula *order by* se a consulta possuir um comando *distinct* ou *group by*, pois a ordenação torna-se redundante;

- Utilizar a cláusula *where* ao invés da *having* junto com o *order by*, pois o *where* elimina as linhas desnecessárias antes que os dados sejam agrupados, enquanto que o *having* elimina essas mesmas linhas depois da agregação;

- Utilizar subconsultas quando a quantidade de dados retornados é pequena ou se não houver índices nas colunas de junção.

No próximo capítulo será apresentada a proposta de solução desenvolvida, destacando o cenário atual do banco de dados, bem como o projeto de banco de dados e o projeto de *tuning* de consulta propostos.

4 PROPOSTA DE SOLUÇÃO

Nas próximas seções serão descritos o cenário atual da base de dados e apresentado o protótipo do projeto de banco de dados criado. Após, será detalhado o projeto de *tuning* de consulta a ser aplicado.

4.1 CENÁRIO ATUAL

A empresa HOS Sistemas de Informática[4], possui sua sede na cidade de Bento Gonçalves. Atua desde 1994, oferecendo soluções para o gerenciamento comercial de farmácias e drogarias, que se adaptam a lojas de pequeno a grande porte, incluindo redes de drogarias. Possui clientes distribuídos em todas as regiões do pais, sendo a maioria localizados no Rio Grande do Sul e Minas Gerais.

Os *softwares*, são desenvolvidos na linguagem *Visual Basic* 6 e integrado com os SGBDs *Firebird* e SQL *Server* 2005. Possui módulos para o gerenciamento completo de farmácias de manipulação e drogarias, incluindo, além do frente de caixa, que será estudado nesse trabalho, módulos de retaguarda, financeiro, fiscal, compras, produtos controlados, convênios e replicador.

A estrutura atual do banco de dados está longe dos conceitos de modelagem de sistema de banco de dados ideal. A primeira base de dados criada, em meados da década de 90, utilizava o Dbase III *Plus*, devido a aplicação ser suportada apenas em DOS. Alguns anos depois, com a migração do *software* para o *Windows*, a base de dados foi convertida para o *Access*, sendo que a estrutura das tabelas e campos foi mantida a mesma. Com o crescimento da empresa e do *software*, o *Access* passou a não suprir mais as necessidades, além de causar grandes transtornos por corromper frequentemente. Sendo assim, decidiu-se pela utilização de outro SGBD: o *Firebird*, onde, mais uma vez, não houve mudanças na estrutura de tabelas e campos, sendo que os dados foram apenas importados de *Access* para *Firebird*. Esse último vem sendo utilizado atualmente, juntamente com o SQL *Server* 2005, em menor escala.

Apesar das diversas migrações de SGBD efetuadas, nunca foi realizado uma reestruturação das tabelas e campos. Sendo assim, herdou-se a estrutura inicial dos arquivos

4 http://www.hos.com.br/

de extensão DBF. A seguir será descrita a estrutura e os problemas enfrentados atualmente com o banco de dados no que se refere a:

- Chave Primária: Essa boa prática é uma das poucas presentes no banco de dados atual, sendo que a maioria das tabelas possui chave primária definida;

- Chave Estrangeira: Apesar da existência de chaves primárias, os conceitos de chave estrangeira não são aplicados. Sendo assim, com a falta de relacionamento entre as tabelas, ocorrem muitos problemas de inconsistência de dados, como por exemplo, ter os registros dos itens vendidos, porém o cabeçalho da venda não existir. A verificação da consistência dos dados, que deve ser uma tarefa do banco de dados, acaba sendo dos desenvolvedores, que quando orientados, realizam esse tipo de verificação ao implementar uma rotina. Outro problema enfrentado, é que muitos relacionamentos são realizados utilizando campos do tipo texto, o que mais uma vez pode causar inconsistência dos dados retornados, quando realizada uma junção;

- Atributos: A grande maioria dos atributos não possui definições de valores *default*, caso o campo não seja de preenchimento obrigatório. Com isso, a prevenção de erros, devido aos campos conterem valor *null*, depende do desenvolvedor realizar a verificação, antes de utilizar o valor do campo para algum processo;

- Índices: Não existe um padrão definido para criação de índices. Geralmente, cria-se índices conforme a necessidade de desempenho que o cliente necessita;

- Visões: O banco de dados possui poucas visões definidas, sendo que essa prática, apesar de proporcionar segurança, não é muito difundida na empresa.

Juntamente com o DBA da empresa, foram elencadas as principais necessidades de melhorias da estrutura do banco de dados atual, sendo elas:

- Necessidade de padronização de nomes de entidades e atributos, incluindo chaves primárias e estrangeiras;
- Utilização de chaves primárias e estrangeiras, bem como padronização de índices em cada tabela;

- Padronização de tipos de dados que serão utilizados, definindo também valores *default* aos atributos de preenchimento não obrigatório;
- Definição de restrições nos relacionamentos, quando realizado exclusão de dados;
- Disseminação do uso de visões, a fim de aumentar a segurança, proporcionando uma visão limitada e controlada dos dados pelos usuários;
- Avaliação da necessidade de utilização de cada entidade e atributo, pois muitos desses não são mais utilizados no *software*.

Nas subseções a seguir, será apresentada as funcionalidades do SQL *Server* 2005 que são utilizadas pela empresa, bem como as funcionalidades do SQL *Server* 2008, que pretende-se utilizar nos novos *softwares* que estão sendo desenvolvidos.

4.1.1 SQL *SERVER* 2005

A exigência do mercado fez com que a empresa procurasse um SGBD mais robusto, que atendesse a clientes mais complexos e exigentes. Sendo assim, optou-se pela integração do *software* com o SQL *Server* 2005[5], na sua versão *Express*, que apesar de gratuita, atende a demanda que a empresa necessita, sendo que a versão suporta banco de dados com tamanho de até 4GB.

O SQL *Server* é desenvolvido pela *Microsoft* para gerenciamento de banco de dados cliente/servidor de forma relacional. Atualmente, possui diferentes edições onde cada uma acomoda desempenho, tempo de execução e requisitos de preço exclusivos para cada tipo de organização. O Anexo A mostra um resumo do processo de evolução e crescimento do SQL *Server*, citando as principais alterações na estrutura do banco de dados ao longo de seus 22 anos (DEWSON, 2008).

O SQL *Server* 2005, mesmo em sua versão *Express*, possui diversas funcionalidades importantes como criptografia dos dados diretamente no banco de dados, armazenamento de dados do tipo XML, criação de tipos de dados personalizados, suporte a procedimentos armazenados, *triggers* e visões (MICROSOFT, 2005). Porém, dentre as funcionalidades destacadas, apenas as três últimas são utilizados atualmente.

O suporte a replicação e *backup* automático são disponíveis apenas em versões pagas do SQL *Server*, porém os mesmos podem ser realizados utilizando outros recursos. Para

5 http://www.microsoft.com/brasil/servidores/sql/2005/default.mspx

replicação de dados, a empresa possui um *software* próprio.

4.1.2 SQL *SERVER* 2008

Em meados do segundo semestre de 2009, a empresa optou em criar um projeto para desenvolvimento de seus *softwares* em uma nova linguagem de programação. Com isso, a escolha pelo SGBD também teve que ser efetuada. Sendo assim, optou-se pela utilização da versão mais recente do SQL *Server*, a 2008[6], em sua edição *Express*. Assim como sua versão anterior, essa última fornece uma plataforma de dados segura, confiável e produtiva que permite executar aplicações de missão crítica, reduzindo o tempo e custo de desenvolvimento e o gerenciamento de aplicações. Por ser considerado um banco de dados robusto, pode ser utilizado por organizações dos mais diversos portes, sendo que as aplicações suportadas vão desde sistemas que utilizam um banco de dados de forma rápida e simples até sistemas de alta complexidade (MICROSOFT, 2010).

Com essa versão do SQL *Server*, pretende-se utilizar as funcionalidades que o SGBD dispõe refere a: procedimentos armazenados; *triggers*, principalmente para uso do *software* de replicação; visões, para algumas rotinas que exigem segurança quanto a visualização de dados; armazenamento de dados do tipo XML, sendo que cada vez mais esse tipo de dado vem sendo utilizado no mercado para envio de dados, e criação de dados personalizados, uma boa prática principalmente para padronização de campos como, por exemplo, descrição, nome, status e valor monetário.

Na seção a seguir será apresentado o protótipo do projeto de banco de dados realizado.

4.2 PROJETO DE BANCO DE DADOS

Sabe-se que a reestruturação de uma base de dados não envolve somente alterar a modelagem de entidades, atributos e relacionamentos, vai além disso, sendo necessário reestruturar, ou até mesmo reconstruir, o *software* que utiliza o banco de dados. Sendo assim, com o projeto de desenvolvimento dos *softwares* em uma nova linguagem de programação, trouxe a oportunidade de modelar a base de dados, aplicando os conceitos de sistema de banco de dados.

6 http://www.microsoft.com/sqlserver/2008/pt/br/default.aspx

No sistema, uma das funcionalidades que mais exige retorno rápido das informações é a de frente de caixa, que envolve operações de venda de produtos e recebimento de contas de clientes e convênio, além de gerar relatório gerenciais, essenciais para o estabelecimento. Essa é uma das funcionalidades que mais requer agilidade. Sendo assim, optou-se por criar um protótipo modelando as entidades, atributos, relacionamentos e índices envolvidos nessas operações. O modelo relacional criado pode ser visualizado no Anexo E, que contempla as entidades referentes a cadastros e o Anexo F, que contempla as entidades referentes a vendas e pagamentos de contas.

Para a criação do protótipo de modelagem do banco de dados foram levados em consideração as necessidades de melhorias sugeridas na seção 4.1, além dos três aspectos a seguir:

- Análise das regras de negócio: Foi realizado um estudo das regras de negócio envolvidas nas operações de frente de caixa. Além das regras atuais do sistema, foram levantadas novas funcionalidades que serão implementadas. Com isso, a base de dados foi modelada para atender essas novas funcionalidades.

- Análise da base de dados atual: Analisando a base de dados atual, juntamente com o DBA, pode-se visualizar de forma clara que dados precisam ser armazenados e como eles devem estar relacionados, para que as regras de negócio envolvidas possam ser contempladas. Além disso, pode-se detectar que várias entidades e atributos não são mais utilizados.

- Aplicação de regras de sistemas de banco de dados: Um dos objetivos do projeto, que esta sendo desenvolvido pela empresa, é poder aplicar as regras de sistemas de banco de dados na modelagem criada. Sendo assim, foram aplicados, na medida do possível, os conceitos e regras de sistema de banco de dados.

4.3 PROJETO DE *TUNING* DE CONSULTA

A aplicação de *tuning* de consulta será desenvolvida em estudos de casos, baseado nos conceitos definidos no capítulo 3. A seguir, serão detalhados como o processo de aplicação de *tuning* será executado.

- Banco de Dados: Serão utilizados dois bancos de dados com diferentes estruturas. O primeiro deles, é o banco de dados atual, contendo apenas as entidades envolvidas nas operações de frente de caixa, citadas na seção anterior. O modelo relacional dessa banco esta apresentado nos Anexos C e D. A segunda base de dados, será criada conforme modelo relacional (Anexos E e F). A importação dos dados, da primeira para a segunda base de dados, será realizada utilizando um importador, que será implementado.

- Técnicas utilizadas: As técnicas de *tuning* de consulta aplicadas, baseiam-se no capítulo 3. Será desenvolvido um estudo de caso para as seguintes técnicas abordadas no capítulo: ordenação, agrupamento, utilização de índices com os predicados *like*, *between* e igualdade (=), e *hints* com aplicação dos *hints* de junções e *index*.

- Escolha das consultas: A escolha das consultas que serão utilizadas para aplicação dos estudos de caso foram definidas juntamente com a equipe responsável pelo projeto na empresa. Foram escolhidas as consultas realizadas com maior frequência e que necessitam de um um bom desempenho.

- Obtenção dos resultados: Para cada estudo de caso realizado, os resultados serão obtidos com o auxilio da ferramenta *Toad for SQL Server*[7], que proporciona uma visão completa dos processos realizados durante a execução das consultas, bem como comparação entre diversas consultas. As técnicas de *tuning* serão aplicadas, nas duas bases de dados, gerando assim comparativos de desempenho entre ambas.

4.4 METODOLOGIA

A metodologia para aplicação de *tuning* tem como base a metodologia de ajuste de desempenho, apresentada na Figura 2, sendo que esta esta adaptada à aplicação dos estudos de caso realizados. A Figura 11 demonstra essa metodologia.

7 http://www.quest.com/toad-for-sql-server/

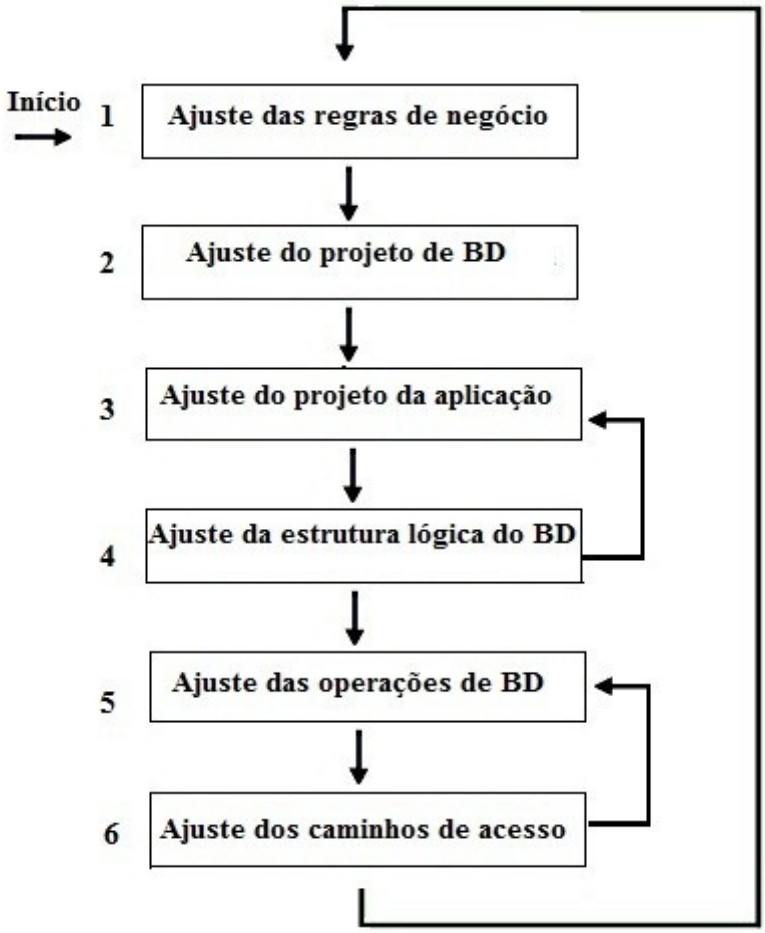


Figura 11: Metodologia de ajuste de desempenho adaptada
Fonte: Autor

4.5 CONSIDERAÇÕES FINAIS

Inicialmente, o capítulo apresentou o cenário atual da empresa, do software desenvolvido pela mesma e, mais pontualmente, da estrutura atual do banco de dados, elencando as principais necessidades de melhoria na estrutura atual. Após, o capítulo apresentou o projeto de *tuning* de banco de dados e de consulta que será aplicado em estudos de casos, tendo como objetivo comparar os resultados obtidos com a aplicação das técnicas de *tuning* abordadas. Finalizando o capítulo, foi apresentada a metodologia adaptada à aplicação de *tuning* desse trabalho.

No próximo capítulo será descrita a implementação da ferramenta desenvolvida para realizar a importação dos dados do banco de dados da estrutura atual para a nova estrutura criada, conforme projeto de banco de dados descrito na seção 4.2 desse capítulo.

5 IMPLEMENTAÇÃO

Para a realização dos estudos de caso, foi necessário implementar uma ferramenta para a importação dos dados da estrutura antiga do banco de dados, conhecida como versão 2.0, para a a versão atual, denominada versão 3.0.

Na próxima seção será apresentada a implementação dessa ferramenta de importação de dados.

5.1 IMPORTADOR

A ferramenta desenvolvida tem como principal funcionalidade realizar a importação dos dados, referentes as entidades relacionadas no Anexo E e Anexo F, do banco de dados na versão 2.0 para a versão 3.0.

Nas próximas subseções serão destacadas as características da ferramenta, *login* principal para execução, adequações realizadas para consistência dos dados, além da visualização do diagrama de classe.

5.1.1 CARACTERÍSTICAS

O banco de dados utilizado como base para os estudos de caso, originalmente encontra-se em *Firebird*. Sendo assim, houve a necessidade de realizar a conversão desse banco de dados para o SQL *Server* 2008, já que comparações entre os bancos de dados nas versões 2.0 e 3.0 serão realizadas. Para a operação de conversão do *Firebird* para o SQL *Server*, foi utilizado um conversor desenvolvido previamente pela empresa que disponibilizou o banco de dados.

A implementação da ferramenta foi realizada com a linguagem de programação C#, utilizando a plataforma de desenvolvimento Visual *Studio* 2008. Basicamente, o projeto de implementação é composto por classes com propriedades que representam as entidades e atributos a serem importados, além de um formulário utilizado como interface de acesso à importação dos dados.

5.1.2 INTERFACE DA FERRAMENTA

Na ferramenta há uma única interface responsável por identificar os bancos de dados que farão parte da importação, bem como demonstrar o progresso da mesma. A Figura 12 demonstra a interface desenvolvida.

Para realizar a importação, deve-se ter os bancos de dados de origem e destino previamente criados no SQL *Server*. Após, é necessário informar nos campos respectivos, o nome dos bancos de dados de origem e destino, sendo possível verificar se a conexão com os bancos de dados informados esta acessível. A execução do importador é iniciada através do botão Importar.

Figura 12: Interface do importador
Fonte: Autor

5.1.3 EXECUÇÃO

Durante o processo de importação, a cada registro, são realizadas consistências, conforme serão destacadas na próxima subseção. No total, são importados em torno de 2.300.000 registros em aproximadamente 4 horas, sendo que esses registros estão divididos em 34 entidades. Ao final da importação, o banco de dados de destino possui um tamanho de 1,25 GB. O processo de importação pode ser acompanhado na interface, apresentada na Figura 12, onde é possível visualizar a quantidade total de registros a serem importados e a quantidade importada até o momento.

No decorrer dos testes da ferramenta, verificou-se uma grande demora na importação de registros da tabela Caixas devido as consistências com a tabela Vendidos. Sendo assim, para diminuir o tempo de importação, foi necessário criar um índice no campo Venda_Id, da tabela Vendidos.

A cada nova importação realizada, todos os dados do banco de dados da versão 3.0 são deletados e inseridos novamente, pois não há possibilidade de selecionar quais tabelas devem ser importadas. A importação é realiza em um ordem lógica de necessidade dos dados.

5.1.4 CONSISTÊNCIAS REALIZADAS

Para ser possível manter os dados da versão 2.0 do banco de dados, os mesmos se tornarem íntegros e possibilitar o relacionamento entre as tabelas, foi necessário realizar algumas consistências nos dados. As consistências implementadas foram:

- Inserção de um registro padrão nos cadastros de apresentação de produtos, ação terapêutica, grupo de produtos, fabricante, colaborador, convênio e clientes, a fim possibilitar a utilização de chaves estrangeiras e não perder o registro, já que a versão 2.0 do banco de dados não possui uma integridade completa dos dados e não tem relacionamento físico entre as tabelas;

- Consistência dos campos do tipo texto, quanto ao tamanho do dado;

- Validação dos registros para atributos que possuem valor único, tanto para chaves primárias quanto para índices únicos;

- Atrelar valores padrões para campos que permitem valores nulos, na versão 2.0 do banco de dados;

- Consistência dos valores registrados nas tabelas Caixas e Vendidos, para que o valor total vendido seja correspondendo ao valor total registrado, nas diferentes formas de pagamento.

5.1.5 DIAGRAMA DE CLASSE

Para uma visualização mais clara da implementação realizada, a Figura 13 demonstra o diagrama de classe da ferramenta implementada. Cada classe implementada representa uma tabela do banco de dados, sendo que cada classe possui propriedades, que representam os atributos da tabela, com os mesmos tipos de dados e relacionamentos da tabela que representa.

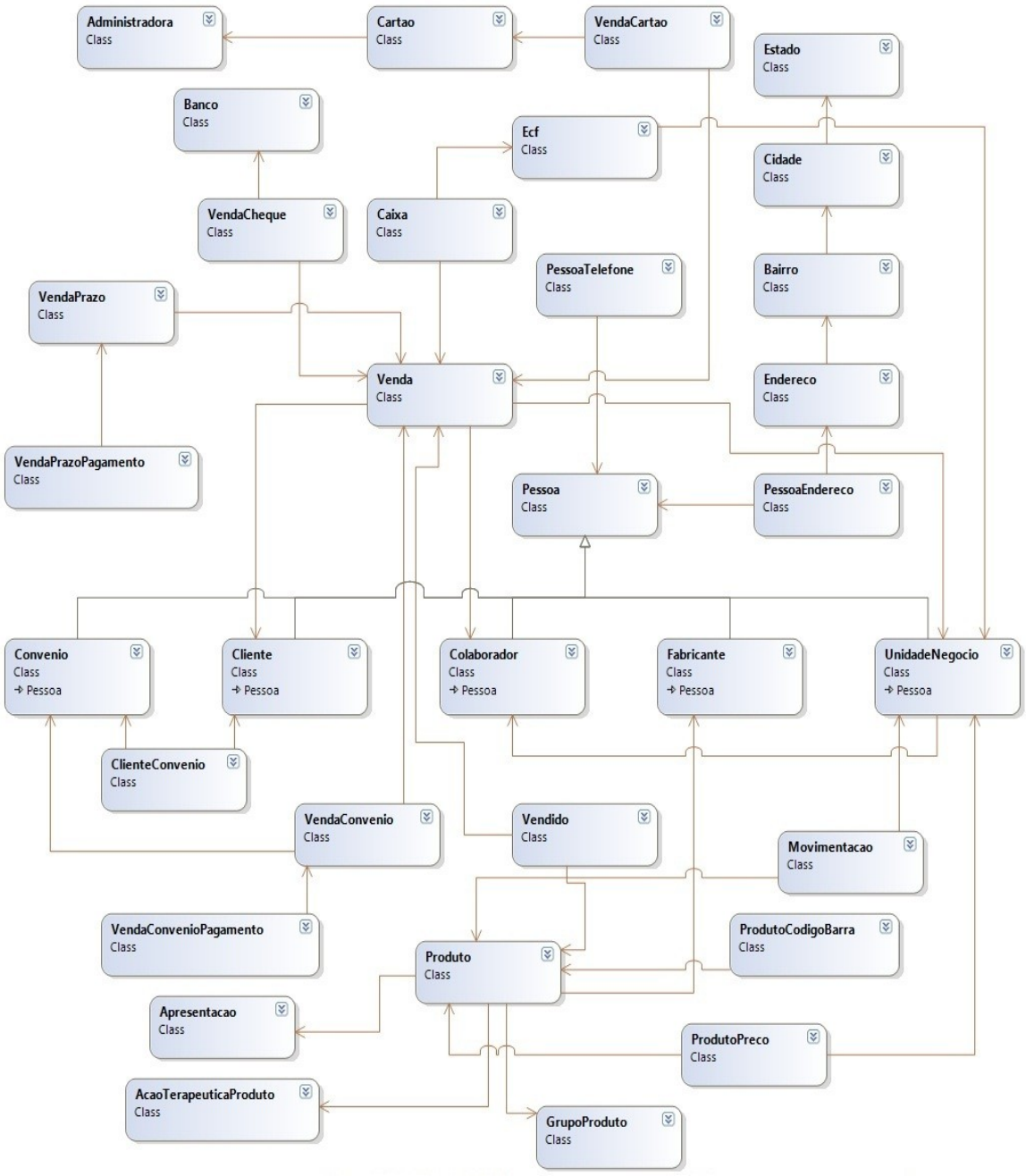


Figura 13: Diagrama de classe do importador
Fonte: Autor

As setas de associação,  , representam os relacionamentos entre as classes e, para representar as heranças da classe Pessoa, é utilizada herança identificada, representada através da seta  . A utilização de herança na implementação foi adotada devido ao fato de haver cadastros de possuem as mesmas características, sendo o caso das classes cliente, representando o cadastro de clientes; colaborador, representando o cadastro de colaboradores; fabricante, representando o cadastro de fabricantes de produtos; unidade de negócio, representando o cadastro das unidades de negócio, e convênio, representando as empresas conveniadas as unidades de negócio.

5.2 CONSIDERAÇÕES FINAIS

No capítulo foi descrito a forma de implementação da ferramenta para importação dos dados da versão 2.0 para a versão 3.0 do banco de dados. Além de demonstrar a interface da ferramenta, aspectos como as características da ferramenta e adequações realizadas para manter a consistência dos dados foram apresentadas no capítulo. Finalizando, foi demonstrado o diagrama de classe, sendo descrito os relacionamentos utilizados entre as classes.

No próximo capítulo será apresentado os estudos de casos realizados, identificando para cada regra de negócio estudada os problemas enfrentados e a solução encontrada, demonstrando dados estatísticos que auxiliam na visualização dos resultados.

6 ESTUDOS DE CASO

Nesse capítulo serão apresentados os estudos de caso da aplicação dos conceitos analisados no capítulo 3 e do projeto de *tuning* de consulta apresentado no capítulo 4.

6.1 CENÁRIO

Os estudos de caso foram realizados em um notebook Dell Inspiron 1545 com processador Intel Core 2 Duo 2.2 Ghz, com 4 GB de memória RAM. Foi utilizado o SGBD SQL *Server* 2008 *Express Edition* e a ferramenta *Toad For Sql Server* para realizar os planos de execução. A estrutura do banco de dados usada está descrita nos Anexos E e F.

Para a obtenção dos resultados de modo a não prejudicar nenhuma análise realizada, a cada consulta executada no SQL *Server*, seu serviço foi encerrado e iniciado novamente.

Para cada estudo de caso que será apresentado, é descrita a regra de negócio que envolve a consulta analisada e identificados os principais problemas enfrentados com cada consulta. Finalizando, é apresentada a análise da solução, incluindo dados estatísticos extraídos com o auxílio da ferramenta *Toad For SQL Server* (FRITCHEY & DAM, 2009; SCHNEIDER & GIBSON, 2008). Os dados estatísticos são:

- Leituras lógicas: informa o número de leituras lógicas executadas, ou seja, o número de páginas lidas em memória.
- Leituras físicas: informa o número de páginas lidas em disco. Se as páginas requeridas por uma consulta não estão na memória devem ser lidas do disco para a memória.
- Páginas lidas por antecipação: são páginas adicionais lidas para efeito de otimização, mantendo-as em *cache* para agilizar sua utilização em outras consultas.
- Escaneamento: informa o número de vezes que as tabelas ou índices foram acessados.
- Tempo total: tempo total para execução da consulta.

Em cada estudo de caso são apresentadas duas consultas, uma que representa a consulta realizada atualmente no sistema, e outra que representa a consulta com aplicação das

técnicas de *tuning*. Com a alteração da estrutura do banco de dados da versão 2.0 para a versão 3.0 as tabelas, campos, relacionamentos e, até mesmo parâmetros envolvidos podem não ser os mesmos entre as duas consultas apresentadas.

As subseções que seguem apresentam os sete estudos de caso realizados que atendem o uso de agrupamento e ordenação, realizam o uso efetivo de índices com uso dos predicados de igualdade, *like* e *between* e, finalizando a aplicação dos três *hints* de junção e o *hint* de tabela *index*.

6.2 ESTUDOS DE CASO 1 – AGRUPAMENTO

O primeiro estudo de caso tem como objetivo demonstrar as técnicas de aplicação de *tuning* referente a agrupamentos.

6.2.1 REGRA DE NEGÓCIO

Para o gerente ou administrador de um estabelecimento comercial é de extrema importância analisar periodicamente o movimento de vendas em seu estabelecimento, analisando principalmente os valores gerados com a comercialização de seus produtos. Sendo assim, deve-se haver a possibilidade de visualizar o montante gerado em cada forma de pagamento para um período pré-definido.

6.2.2 PROBLEMA

Um dos grandes problemas que envolve uma consulta com essas características é realizar o agrupamento dos dados devido a grande quantidade de registros envolvidos, sendo que, no banco de dados analisado, há uma média de 800 registros de vendas por dia.

6.2.3 SQL ORIGINAL

A consulta original, que pode ser visualizada abaixo, é composta apenas pela tabela Caixa, sendo que nela há índices não clusterizados para cada atributo da cláusula *where* da

consulta. A Figura 14 mostra o plano de execução da consulta original.

Nessa tabela são armazenadas os registros de movimentação de caixa da farmácia. Além das vendas, são registradas devoluções e recebimento de contas a prazo e convênios. Mas, para o estudo de caso em questão, são necessários apenas os registros de vendas que são restritos aos lançamentos de cheque (CH), cheque pré-datado (CP), cartão (CR), venda convênio (VC), venda a prazo (VP) e venda a vista (VV).

```
SELECT c.Data, sum(c.Dinheiro) as dinheiro, sum(c.Cheque) as cheque, sum(c.Cartao) as cartao,
        sum(c.Aprazo) as aprazo, sum(c.Convenio) as convenio, sum(c.Outros) as outros,
        sum(c.Cheque_pre) as cheque
        FROM Caixa c
WHERE c.Lancamen in('CH','CP','CR','VC','VP','VV')
                and c.Data between '01/01/2010' and '01/31/2010'
                and c.Empresa = 1
GROUP BY  c.Data
ORDER BY  c.Data
```

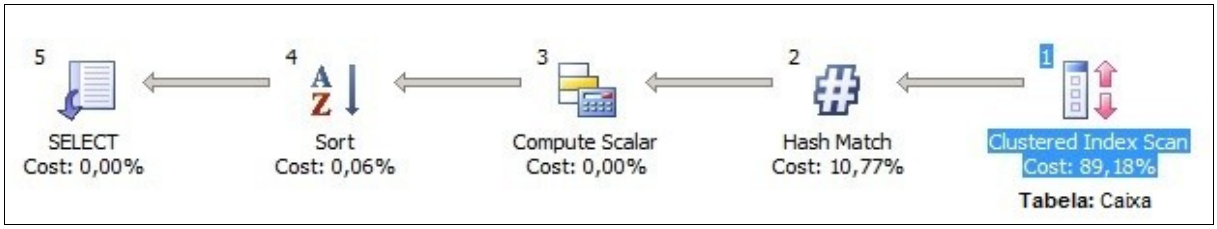


Figura 14: Plano de execução da consulta original - Estudo de caso 1
Fonte: Autor

De acordo com o plano de execução gerado, inicialmente foi acionado o índice da chave primária da tabela Caixa para realizar a seleção dos registros que pertencem a cláusula *where*. Em seguida, os campos são selecionados e foi realizada a agregação dos valores. Finalizando, os registros são ordenados e a projeção dos dados é realizada.

6.2.4 SOLUÇÃO

Conforme citado no capítulo 3, segundo Fritchey & Dam (2009) para otimizar o uso do agrupamento recomenda-se criar um índice com os atributos que fazem parte da cláusula *group by* e os atributos de agregação. Sendo assim, foi criado na tabela Caixas um índice

composto pelos atributos de agregação, chamado de Idx_FormaPagamento, e na tabela Vendas, um índice composto pelos atributos presentes na cláusula *where*, denominado de Idx_DataUnidadeNegocio. Além disso, a consulta foi reescrita, seguindo algumas considerações sobre junções apresentadas no capítulo 3.

Na versão 3.0 do banco de dados, os campos que armazenam datas/horas são do tipo *datetime*, diferente da versão 2.0 onde havia um campo para armazenar data e outro para hora. Sendo assim, com a necessidade de agrupamento através da data, foi necessário aplicar a função *convert* do SQL Server, que converte uma expressão de um tipo de dado para outro (MSDN, 2009). Os parâmetros dessa função representam, nessa ordem, o tipo de dado após a conversão, o campo a ser convertido e o formato de apresentação do dado após a conversão. Na consulta abaixo, o campo DataHora será convertido para um tipo texto no formato 103, que representa dd/mm/aaaa.

Como pode ser visualizado na consulta a seguir, as restrições não são as mesmas. A restrição do tipo de lançamento do registro, explicada na seção anterior, não é mais utilizada pois com a versão 3.0 a tabela Caixas passa a armazenar somente registros de vendas. Outro campo que possui o parâmetro alterado é o campo que representa a unidade de negócio, sendo que após a conversão dos dados, o registro que representa a unidade de negócio 1 para a ser representado pelo valor 586B3FF2-1230-43C9-A557-114319E672EC que identifica a chave do registro na tabela UnidadesNegocios. A Figura 15 apresenta o plano de execução dessa consulta.

```
SELECT convert(nvarchar(10), v.DataHora,103) as Data, sum(c.ValorDinheiro) as valorDinheiro,
        sum(c.ValorCartao) as valorCartao, sum(c.ValorCheque) as valorCheque,
        sum(c.ValorConvenio) as valorConvenio, sum(c.ValorChequePre) as valorChequePre,
        sum(c.ValorAPrazo) as ValorAPrazo, sum(c.ValorOutros) as valorOutros
        FROM Vendas v
        INNER JOIN Caixas c on v.Id=c.Venda_Id
WHERE v.DataHora between '01/01/2010' and '01/31/2010'
        and v.UnidadeNegocio_Id='586B3FF2-1230-43C9-A557-114319E672EC'
GROUP BY convert(nvarchar(10), v.DataHora,103)
```

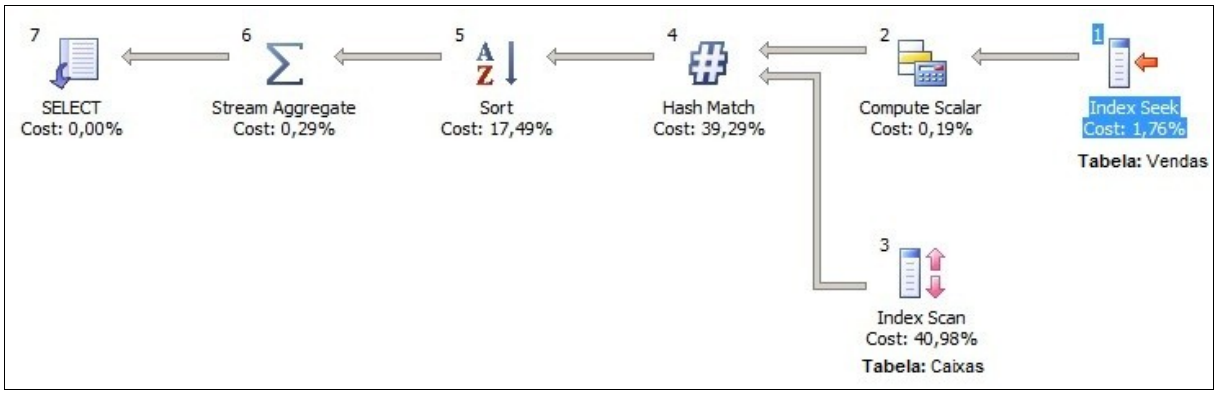


Figura 15: Plano de execução da consulta com *tuning* - Estudo de caso 1
Fonte: Autor

Após a reescrita da consulta a mesma passa a envolver duas tabelas. Isso se deve ao fato da mudança na estrutura das tabelas envolvidas, Caixas e Vendas, sendo que a tabela Caixas passa a não ter mais colunas que identifiquem a data e a empresa, sendo assim é necessário realizar a junção com a tabela Vendas para obter essa informação.

Na parte superior do plano de execução, os dois primeiros passos referem-se a tabela Vendas, onde é realizada uma busca por *index seek*, utilizando o índice Idx_DataUnidadeNegocio, aos registros que contemplam a cláusula *where* e convertido o campo DataHora no formato especificado. Na parte inferior, no terceiro passo, é realizada a busca por *index scan*, utilizando o índice Idx_FormaPagamento, aos atributos de agregação da tabela Caixas. Após, é realizada a junção das tabelas com o método de junção *hash match* e, finalizando, os registros são ordenados, agrupados e a projeção dos dados é realizada.

Com as alterações acima citadas, é possível identificar, através dos gráficos dispostos na Figura 16 e através da Tabela 4, que a consulta com aplicação de *tuning* foi mais eficiente.

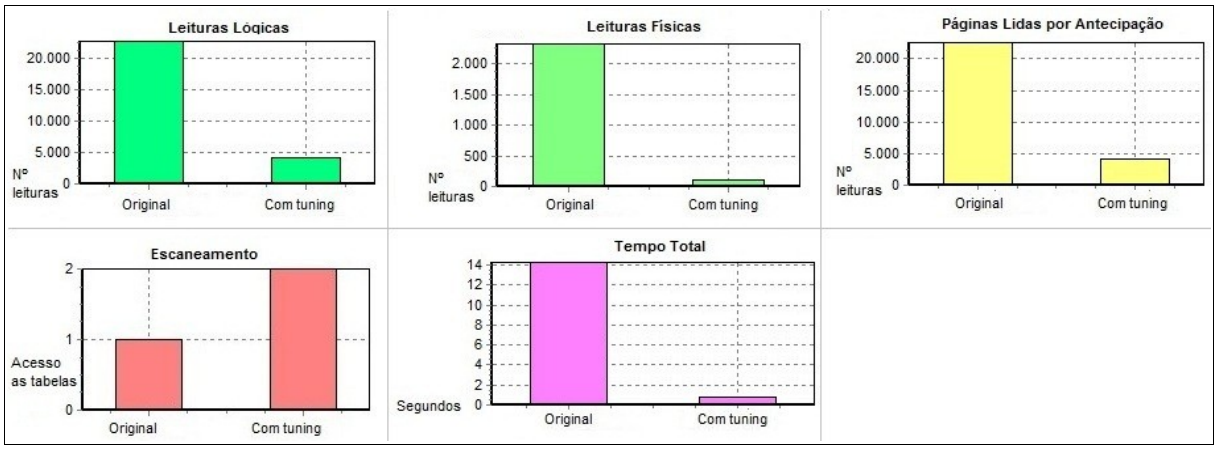


Figura 16: Gráficos comparativos - Estudo de caso 1
Fonte: Autor

Tabela 4: Dados estatísticos das consultas – Estudo de caso 1
Fonte: Autor

Informação Analisada	SQL Original	SQL com *tuning*
Leituras lógicas	22700	4197
Leituras físicas	2331	106
Páginas lidas por antecipação	22564	4190
Escaneamento	1	2
Tempo total	14,29	0,8

A criação dos índices nas tabelas Vendas e Caixas justifica a expressiva queda de leituras lógicas e físicas apresentadas na SQL com *tuning*, sendo assim o tempo total de execução da consulta diminuiu drasticamente. O único aumento que pode ser visualizado refere-se ao *scan*, mas isso se deve ao fato que na SQL original há apenas uma tabela, enquanto que na SQL com *tuning* estão envolvidas duas tabelas e cada uma teve um acesso.

6.3 ESTUDO DE CASO 2 – ORDENAÇÃO

O segundo estudo de caso tem como objetivo demonstrar as técnicas de aplicação de *tuning* referente a ordenação de consultas.

6.3.1 REGRA DE NEGÓCIO

A satisfação do cliente em encontrar o que precisa e com preços atrativos garante a sobrevivência em um mercado competitivo, como é o mercado do ramo farmacêutico. Uma das maneiras de garantir a sobrevivência é gerando promoções dos produtos que apresentam maior número de vendas. Sendo assim, deve-se ter a possibilidade de visualização dos produtos vendidos em um determinado período de tempo.

6.3.2 PROBLEMA

Apesar da consulta não ser executada com frequência, alguns fatores podem interferir no seu desempenho, como por exemplo, a ordenação dos resultados. Em alguns casos, o ideal

é retornar os registros já ordenados para facilitar a visualização e encontrar os registros que deseja com mais agilidade. Porém, sabe-se que a ordenação possui um alto custo. É possível retornar os registros ordenados sem que se utilize a cláusula *order by*.

6.3.3 SQL ORIGINAL

A consulta original é composta por duas tabelas: Vendidos e Produtos. Dos campos utilizados na consulta abaixo referente a tabela Vendidos, apenas o campo Data possui um índice não clusterizado e, na tabela Produtos, apenas o campo Codigo, que é a chave primária da tabela, possui um índice clusterizado. A Figura 17 representa o plano de execução da consulta.

```
SELECT p.Descricao, count(vend.Produto)
        FROM Vendidos vend
         INNER JOIN Produtos p   on vend.Produto = p.Codigo
WHERE vend.Data between '09/01/2009' and '09/30/2009'
GROUP BY p.Descricao
ORDER BY p.Descricao
```

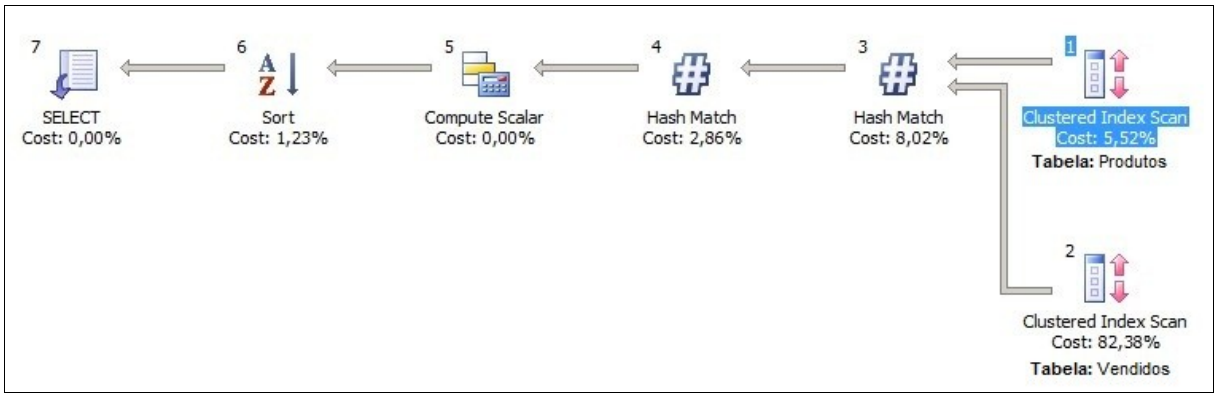


Figura 17: Plano de execução da consulta original - Estudo de caso 2
Fonte: Autor

Para a seleção dos registros especificados na cláusula *where*, é realizada uma busca utilizando o índice clusterizado da chave primária da tabela Produtos, representado pelo primeiro passo do plano de execução. A seguir, é realizada a pesquisa dos registros na tabela Vendidos, com o auxilio do índice clusterizado dessa tabela. Seguindo, é realizada a junção

entre as tabelas e o agrupamento dos registros. Finalizando, é executada a função *count* e ordenado os registros.

6.3.4 SOLUÇÃO

Conforme apresentado no capítulo 3, deve-se evitar o uso da cláusula *order by* quando a consulta possui *distinct* ou *group by*, pois a operação de ordenação torna-se redundante (GULUTZAN & PELZER, 2003). Para a reescrita da consulta, além de considerar a versão 3.0 do banco de dados, foram aplicadas as considerações acima abordadas.

Além de remover a cláusula *order by* da consulta, foi criado um índice não clusterizado Idx_VendaProduto na tabela Vendidos, composto pelos campos Venda_Id e Produto_Id. Esse índice auxilia no desempenho da junção com as tabelas Vendas e Produtos. A consulta reescrita é apresentada a seguir, juntamente com seu plano de execução, apresentado na Figura 18.

```
SELECT p.Descricao,  count(vend.Produto_id)
        FROM Vendas v
       INNER JOIN Vendidos vend  on v.Id = vend.Venda_Id
       INNER JOIN Produtos p  on vend.Produto_Id = p.Id
WHERE v.DataHora between '09/01/2009' and '09/30/2009'
GROUP BY p.Descricao
```

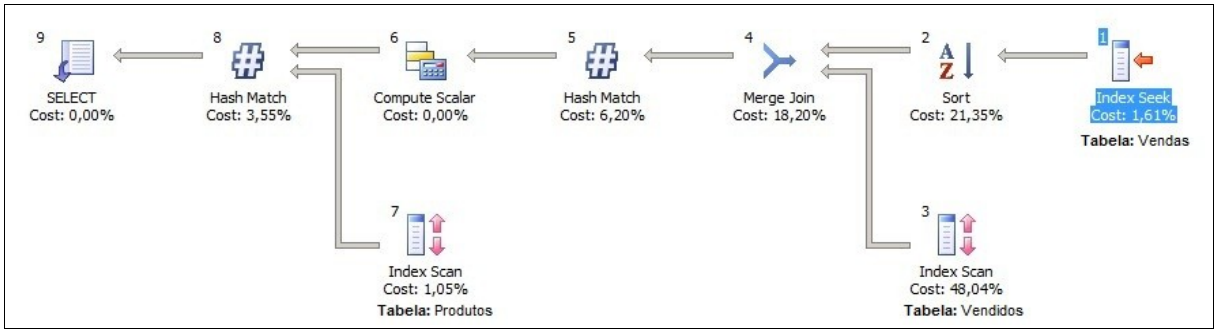


Figura 18: Plano de execução da consulta com *tuning* - Estudo de caso 2
Fonte: Autor

Devido a reestruturação da base de dados para a versão 3.0, a consulta passa a envolver três tabelas: Vendas, Vendidos e Produtos, pois a tabela Vendidos não possui mais o campo que identifica a data da venda, sendo assim necessário realizar uma junção com a

tabela Vendas para que sejam selecionados os registros conforme o período requerido.

Assim como no plano de execução da consulta original, o primeiro passo executado é a busca pelos registros que contemplam a cláusula *where*, porém nesse caso, é utilizado o índice não clusterizado Idx_DataUnidadeNegocio, da tabela Vendas, através da operação de *index seek*. O segundo passo, apesar de não haver uma cláusula *order by* na consulta, realiza a ordenação dos registros considerando o campo da chave do índice clusterizado da tabela Vendas. Já para a seleção dos registros referentes a tabela Vendidos, é utilizado o índice não clusterizado Idx_VendaProduto. Seguindo, é realizada a junção entre as tabelas Vendas e Vendidos utilizando o método de *merge join*. O próximo passo executa a função de *count* com os registros previamente selecionados da tabela Vendidos e, junto com os registros da tabela Produtos, pesquisados como o auxílio do índice clusterizado dessa tabela, executam a consulta.

A seguir são apresentados os gráficos, dispostos na Figura 19, comparando os dados estatísticos da consulta original com a consulta com *tuning*. Na Tabela 5 é possível identificar os resultados obtidos com a comparação.

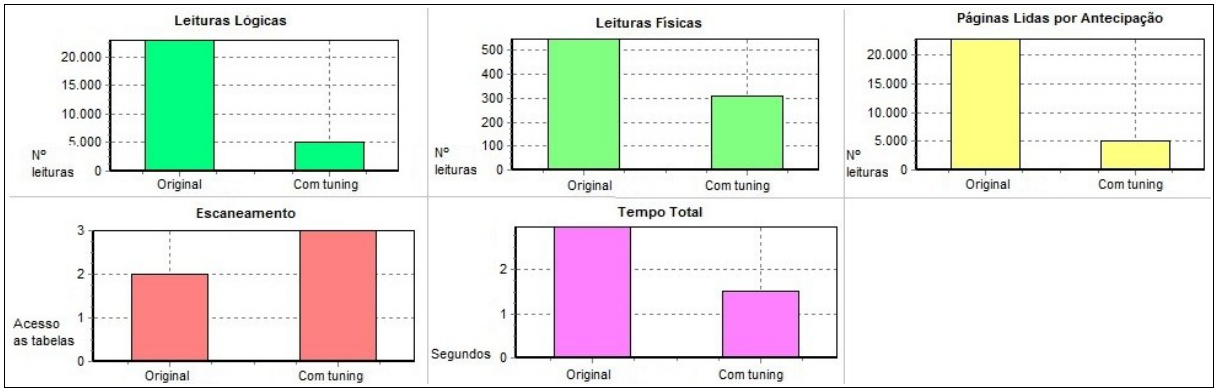


Figura 19: Gráficos comparativos - Estudo de caso 2
Fonte: Autor

Tabela 5: Dados estatísticos das consultas – Estudo de caso 2
Fonte: Autor

Informação Analisada	**SQL Original**	**SQL com *tuning***
Leituras lógicas	22912	4871
Leituras físicas	547	289
Páginas lidas por antecipação	22833	4865
Escaneamento	2	3
Tempo total	3,42	1,78

Apesar da SQL com *tuning* utilizar três tabelas, tendo assim que realizar duas junções, seu desempenho foi melhor. A exclusão da cláusula *order by* da consulta e a criação do índice Idx_VendaProduto, na tabela Vendidos, foram fundamentais para que o tempo total de execução da consulta diminuísse pela metade. Além disso, a leituras lógicas e físicas também apresentam uma queda significativa.

6.4 ESTUDO DE CASO 3 – USO DE ÍNDICES COM OPERADOR DE IGUALDADE

O terceiro estudo de caso tem como objetivo demonstrar as técnicas de uso efetivo de índices, utilizando o operador de igualdade (=) na cláusula *where*.

6.4.1 REGRA DE NEGÓCIO

Ter o controle sobre o estoque dos medicamentos faz com que a farmácia economize no momento da reposição do seu estoque, pois adquire apenas o que necessita e também gere lucros, pois não há falta do produto para comercialização. Sendo assim, a pesquisa pelo estoque dos medicamentos é uma das consultas mais executadas durante o dia a dia da farmácia, sendo que grande parte dessas consultas são realizadas nas operações de venda.

6.4.2 PROBLEMA

Devido à frequência com que a consulta de estoque dos medicamentos é solicitada, o retorno dos resultados desejados deve ser o menor possível para, principalmente, não prejudicar o atendimento realizado na farmácia.

A tabela que armazena esses dados, além de ser utilizada em milhares de consulta de estoque, também recebe uma grande quantidade de inserções. Na base de dados analisada, tem-se uma média de 2400 inserções por dia.

6.4.3 SQL ORIGINAL

A consulta original, mostrada a seguir, é composta apenas pela tabela Movimentacao, sendo que nela há um índice não clusterizado composto pelos dois atributos da cláusula *where* e também outros dois índices criados para os atributos Produto e Empresa, separadamente. Nessa consulta será retornado o estoque do produto com código 75418 na empresa 1. A Figura 20 mostra o plano de exceção dessa consulta.

```
SELECT sum(m.Entrada-m.Saida) as Estoque
        FROM Movimentacao m
WHERE m.Produto=75418 and m.Empresa=1
```

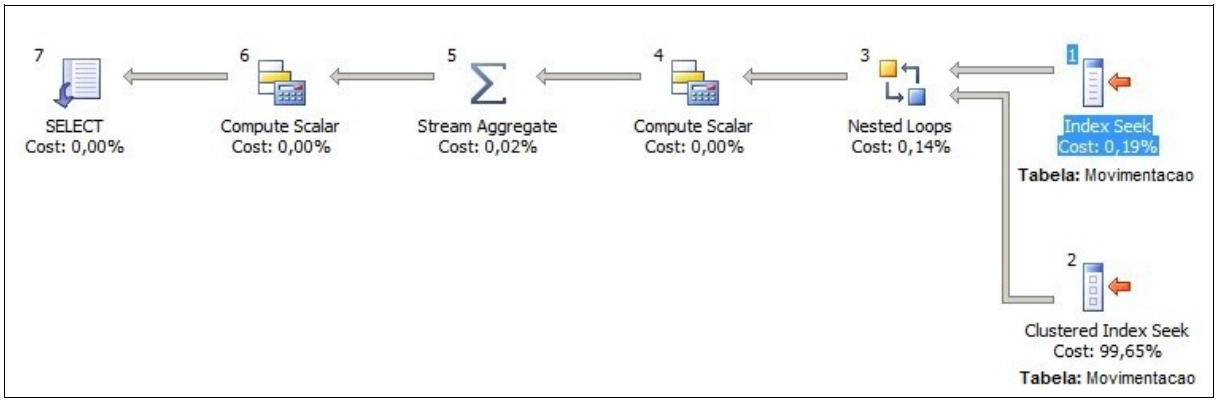


Figura 20: Plano de execução da consulta original – Estudo de caso 3
Fonte: Autor

Conforme o plano de execução, os dois primeiros passos são os índices para seleção dos registros: o primeiro composto pelos campos Produto e Empresa, e o segundo é índice da chave primária da tabela Movimentacao. Em seguida, é realizada a pesquisa utilizando esses índices. O próximo passo, realiza a operação Entrada – Saida e após, a agregação dos registros. Finalizando, a função *count_big* é executada internamente e a projeção dos dados é realizada. Essa função, de acordo com o MSDN (2009) "retorna o número de itens de um grupo". Sendo assim, se a agregação não retornar nenhum registro o resultado apresentado é *null*.

6.4.4 SOLUÇÃO

O predicado de igualdade, conforme a Tabela 1 apresentada no capítulo 3, é considerado um predicado do tipo *sargable*, ou seja, que faz uso de índices quando utilizado na cláusula *where* das consultas. Porém, deve-se sempre analisar quais campos farão parte do índice.

Na consulta em questão, que tem relacionada apenas a tabela Movimentacoes, foram realizadas duas alterações, além da reescrita da consulta. A primeira alteração é a inserção do campo computado Estoque que realiza o cálculo Entrada – Saida para cada registro. A segunda alteração, foi a criação do índice Idx_Movimentacao composto pelos campos Produto_Id, UnidadeNegocio_Id, que compõem a cláusula *where*, e pelo campo Estoque, utilizando para realizar a agregação dos registros. Os campos são apresentados no índice nessa na mesma ordem que foram citados.

Para a criação do índice foram analisadas duas alternativas: o índice ser composto por todos os campos presentes na consulta, ou seja, os campos Produto_Id, UnidadeNegocio_Id e Estoque, ou ser composto apenas pelos campos Produto_Id e UnidadeNegocio_Id. Após análise do plano de execução com as duas situações citadas, pode-se comprovar que a inserção do campo Estoque, além dos campos Produto_Id e UnidadeNegocio_Id, no índice faz com que a consulta apresente uma melhora considerável no tempo de execução, pois utiliza efetivamente o índice criado. De modo contrário, quando é criado o índice apenas com os campos Produto_Id e UnidadeNegocio_Id, o índice utilizado na consulta é o da chave primária da tabela, conforme pode ser visualizado no plano de execução apresentado na Figura 21.

Apesar da tabela receber centenas de registros diariamente, a criação do índice Idx_Movimentacao não alterou de forma significativa o tempo para executar as inserções. Em testes realizados, para inserir blocos de 2000, 50 e 1 registro os tempos apresentados são os demonstrados na Tabela 6.

Tabela 6: Tempos para inserção de registros na tabela Movimentacoes

Quantidade de registros	Inserção sem índice	Inserção com índice
2000	Aproximadamente 28 segundos	Aproximadamente 39 segundos
50	Aproximadamente 2 segundos	Aproximadamente 3 segundos
1	Aproximadamente 2 segundos	Aproximadamente 2 segundos

A seguir é apresentada a consulta reescrita que apresenta alterações nos valores das restrições impostas. O campo que representa o código do produto para a ser representado pelo valor 7BE5B573-E4E0-4445-A46B-F71AF68D8A4F que identifica a chave do registro na tabela Produtos. O campo UnidadeNegocio_Id segue a explicação já abordada no estudo de caso 1. Na figura 22, é apresentado o plano de execução da consulta utilizando o índice composto pelos três campos envolvidos na consulta.

```
SELECT sum(m.Estoque) as Estoque
        FROM Movimentacoes m
WHERE m.Produto_Id='7BE5B573-E4E0-4445-A46B-F71AF68D8A4F'
        and m.UnidadeNegocio_Id='586B3FF2-1230-43C9-A557-114319E672EC'
```

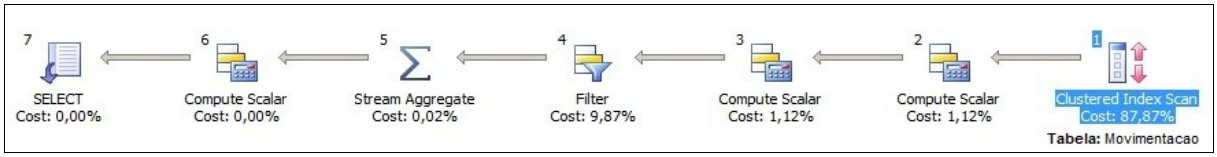


Figura 21: Plano de execução da consulta com índice nos campos Produto_Id e UnidadeNegocio_Id
Fonte: Autor

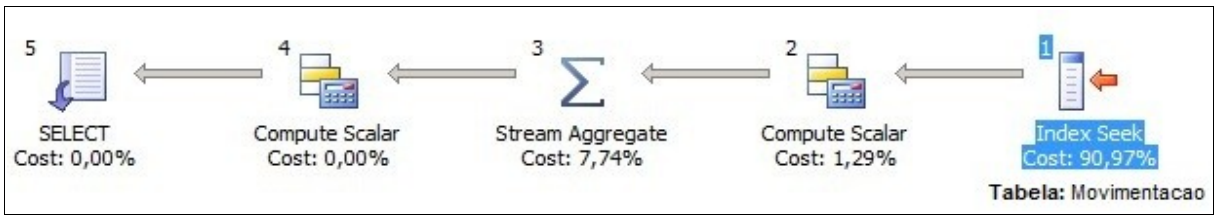


Figura 22: Plano de execução da consulta com *tuning* – Estudo de caso 3
Fonte: Autor

Os planos de execução apresentados para as duas análises de índices são diferentes, não só no tipo de pesquisa utilizada para o índice. No primeiro plano, após a busca pelos registros, é realizada a operação entre os campos Entrada e Saida, retornando o valor para o campo Estoque. Seguindo, os registros são filtrados conforme as restrições impostas na consulta. Finalizando, é realizada a agregação entre todos os registros, aplicada a função *count_big* e projetados os dados.

Já no segundo plano de execução, a consulta continua utilizando apenas uma tabela para obtenção do estoque dos produtos, porém o plano de execução passa a ser menor em número de passos realizados, isso se deve principalmente a utilização efetiva do índice.

Como pode ser visualizado, no primeiro passo do plano de execução o tipo de índice utilizado foi do tipo *index seek*, que lê somente as informações necessárias, deixando a consulta mais eficiente (RIBEIRO, 2004). Após a leitura dos registros, é realizada a agregação dos valores, aplicada a função *count_big* e retornado os registros desejados.

A seguir são apresentados os gráficos, dispostos na Figura 23, bem como os resultados apresentados na Tabela 7, comparando a consulta original e a consulta utilizando o índice composto pelos campos Produto_Id, UnidadeNegocio_Id e Estoque.

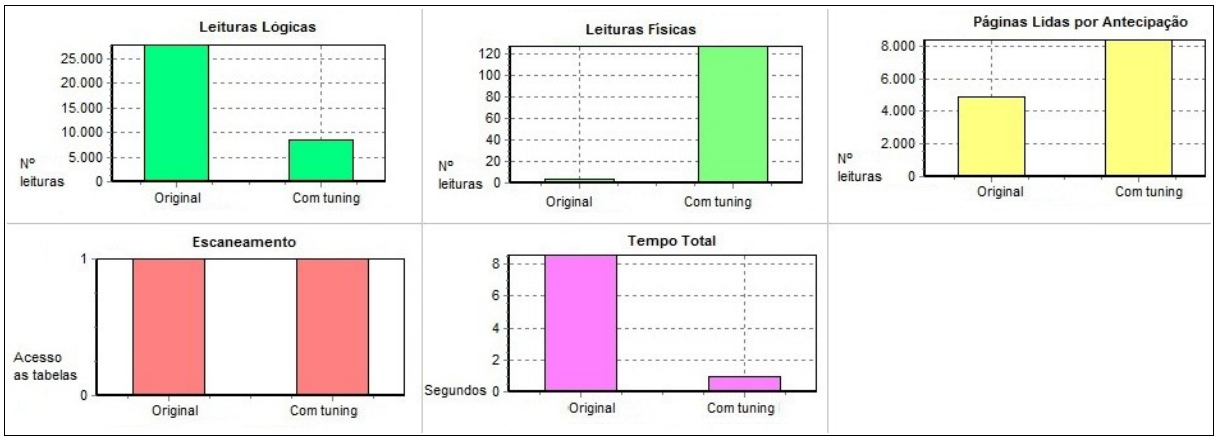


Figura 23: Gráficos comparativos - Estudo de caso 3
Fonte: Autor

Tabela 7: Dados estatísticos das consultas – Estudo de caso 3
Fonte: Autor

Informação Analisada	**SQL Original**	**SQL com *tuning***
Leituras lógicas	27708	8399
Leituras físicas	4	127
Páginas lidas por antecipação	4876	8370
Escaneamento	1	1
Tempo total	8,66	1,1

A consulta com *tuning* apresentou melhora significativa no tempo de execução, além de diminuir o número de leituras lógicas. Essa diminuição deve-se ao fato da utilização do *index seek*, que realiza uma busca pontual e específica. Porém, houve aumento das leituras físicas, sendo que esse aumento esta relacionado a operação física *stream aggregate* que, comparada a consulta original, tem um custo significativamente mais elevado. Para realizar a agregação dos valores, o operador físico necessita acessar os registros na tabela, gerando assim leituras em disco.

6.5 ESTUDO DE CASO 4 – USO DE ÍNDICES COM OPERADOR *LIKE*

O objetivo desse estudo de caso é demonstrar as técnicas de uso efetivo de índices, utilizando o operador *like* na cláusula *where*.

6.5.1 REGRA DE NEGÓCIO

Para a realização de vendas com identificação de cliente é necessário realizar a busca pelo cliente, sendo que essa busca, na maioria das vezes em que é executada, é composta por parte do nome do cliente. A consulta, além de retornar o nome do cliente, deve verificar se o cliente esta apto a realizar a compra, considerando para isso apenas os clientes ativos.

6.5.2 PROBLEMA

A consulta de clientes no ato da venda possui uma frequência de execução considerável. Na base de dados analisada do total de vendas registradas, 30% são de vendas com identificação do cliente. Atualmente a consulta possui um baixo custo de execução, mas esse custo pode ser diminuído ainda mais.

6.5.3 SQL ORIGINAL

A consulta original, apresentada a seguir, é composta apenas pela tabela Clientes, sendo que nela há apenas um índice clusterizado referente a chave primária da tabela. A Figura 24 mostra o plano de execução dessa consulta.

```
SELECT  c.Nome
        FROM Clientes c
WHERE c.Nome LIKE 'SAB%' and c.Status='ATIVO'
ORDER BY c.Nome
```

Figura 24: Plano de execução da consulta original - Estudo de caso 4
Fonte: Autor

O plano de execução gerado é bem simples possuindo apenas três passos. O primeiro deles examina o índice clusterizado da tabela Clientes retornando apenas os registros que contemplam a cláusula *where*. Após, é realizada a ordenação dos resultados e por fim, retornada a consulta.

6.5.4 SOLUÇÃO

O predicado *like* no formato <literal>%, conforme a Tabela 1 apresentada no capítulo 3, assim como o predicado de igualdade apresentado no estudo de caso anterior, também é considerado um predicado do tipo *sargable*.

Para otimizar a consulta foi criado, na tabela Pessoas, um índice não clusterizado Idx_NomeStatus composto pelos campos Nome e Status, além de reescrever a consulta conforme a estrutura da versão 3.0 do banco de dados. Com essa versão, o parâmetro do campo Status, que na consulta original é do tipo texto, passa a ser do tipo binário. Sendo assim, o plano de execução da consulta passa a ser o apresentado na Figura 25.

```
SELECT p.Nome
        FROM Pessoas p
        INNER JOIN Clientes c on p.Id=c.Pessoa_Id
WHERE p.Nome LIKE 'SAB%' and p.Status=1
ORDER BY p.Nome
```

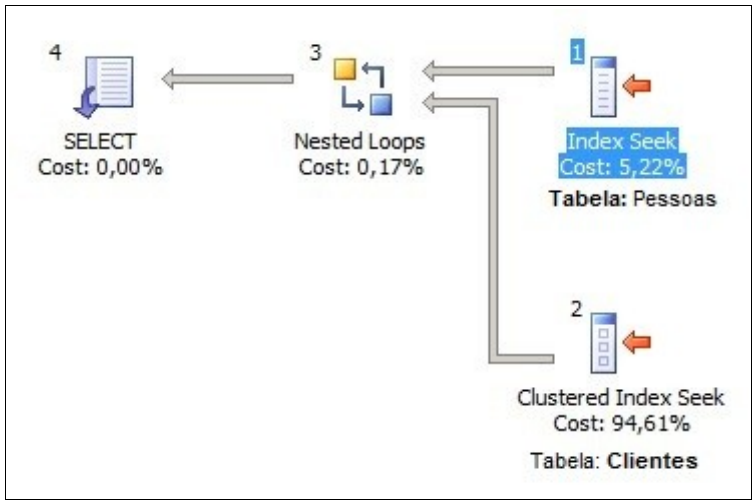


Figura 25: Plano de execução consulta com *tuning* - Estudo de caso 4
Fonte: Autor

Após a reescrita da consulta a mesma passa a realizar uma junção entre as duas tabelas envolvidas. Isso se deve a mudança da estrutura do banco de dados, que inclui a tabela Pessoas para armazenar os dados comuns a uma pessoa. Sendo assim, a tabela Clientes não contém os campos Nome e Status fazendo-se necessário realizar a junção entre as tabelas Pessoas e Clientes.

Os primeiros passos do plano de execução utilizam, respectivamente, o índice Idx_NomeStauts, da tabela Pessoas, e o índice Pk_Cliente, da tabela Clientes, para realizar a busca pelos registros que contemplam a cláusula *where* da consulta. Após, é executa a junção entre as duas tabelas e retornados os registros.

A seguir são apresentados os gráficos, dispostos na Figura 26, comparando as duas consultas, bem como os resultados apresentados na Tabela 8.

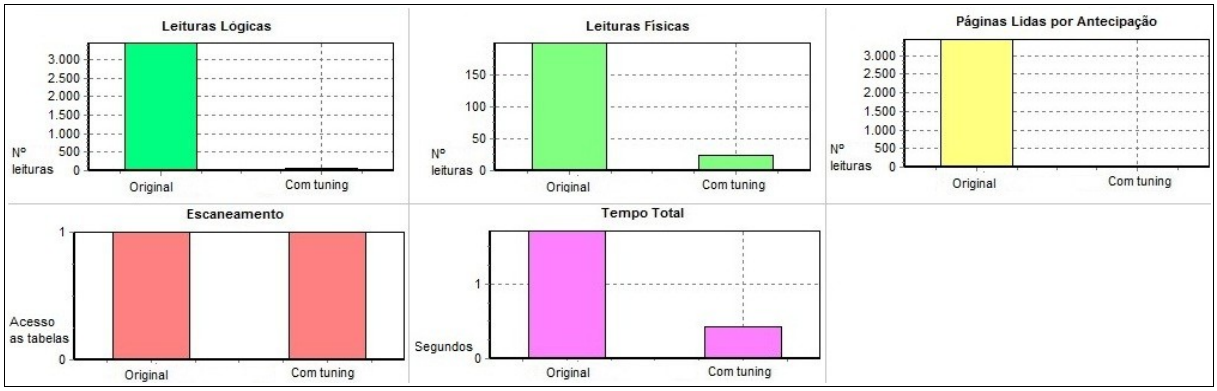


Figura 26: Gráficos comparativos - Estudo de caso 4
Fonte: Autor

Tabela 8: Dados estatísticos das consultas – Estudo de caso 4
Fonte: Autor

Informação Analisada	SQL Original	SQL com *tuning*
Leituras lógicas	3456	72
Leituras físicas	199	21
Páginas lidas por antecipação	3439	0
Escaneamento	1	1
Tempo total	1,78	0,42

A inserção do índice Idx_NomeStatus, além de diminuir todos os dados estatísticos, também reduziu o tempo total de execução, conseguindo alcançar o objetivo da aplicação de *tuning* nessa consulta. Além disso, como pode ser visualizado na Figura 25, a consulta utiliza um *index seek* para realizar a pesquisa na tabela Pessoas. Se a consulta fosse realizada sem o índice, seria executada com *clustered index scan*, que apesar de utilizar o índice clusterizado, realiza um *scan* na tabela o que torna a consulta mais demorada.

6.6 ESTUDO DE CASO 5 – USO DE ÍNDICES COM O OPERADOR *BETWEEN*

O quinto estudo de caso tem como objetivo demonstrar o uso efetivo de índices utilizando o operador *between* na cláusula *where*.

6.6.1 REGRA DE NEGÓCIO

No mundo dos negócios, um dos objetivos é sempre faturar mais e, no ramo farmacêutico essa regra também se aplica. Uma das funcionalidades que auxilia no aumento das vendas é a possibilidade de consulta do histórico de produtos já adquiridos pelos clientes em vendas anteriores, assim o atendente pode oferecer esses produtos no ato da venda, podendo assim contribuir com o aumento das mesmas. Além disso, essa atenção que o estabelecimento tem o cliente pode garantir a fidelidade do cliente a aquela farmácia.

6.6.2 PROBLEMA

Por se tratar de uma consulta realizada enquanto o cliente é atendido, ela deve ser executada o mais rápido possível para que o atendente identifique os produtos e ofereça os mesmos em um curto espaço de tempo.

6.6.3 SQL ORIGINAL

Apesar da consulta retornar somente o nome dos produtos, para que se chegue a esse resultado é preciso unir três tabelas: Vendas, Vendidos e Produtos. Dos campos envolvidos na consulta, os campos Codigo, da tabela Produtos e os campos Vendas e Empresa, da tabela Vendas, possuem índice clusterizado, e o campo Data, também da tabela Vendas possui um índice não clusterizado.

Abaixo é apresentada a consulta original que filtra entre um período de data específico os produtos comprados pelo cliente com código 62660. A Figura 27 mostra o plano de execução dessa consulta.

```
SELECT distinct(p.Descricao) FROM Vendas v
        INNER JOIN Vendidos vend on v.Venda= vend.Venda and v.Empresa = vend.Empresa
        INNER JOIN Produtos p on vend.Produto= p.Codigo
WHERE  v.Data BETWEEN '01/01/2010' and '03/31/2010' and v.Cliente=62660
```

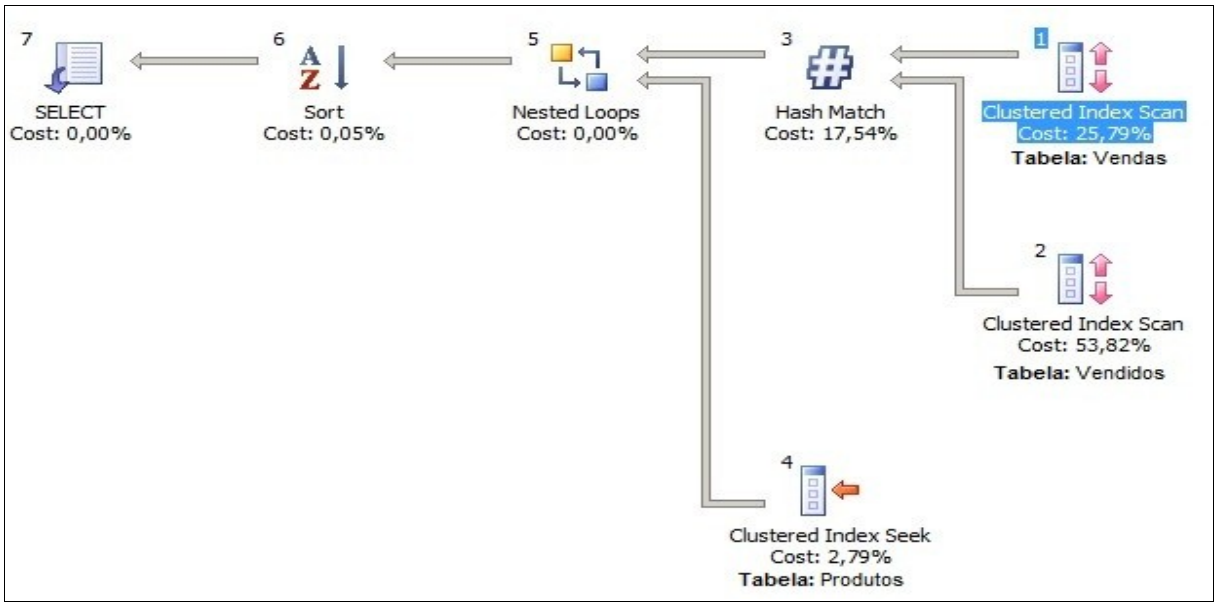


Figura 27: Plano de execução da consulta original - Estudo de caso 5
Fonte: Autor

O plano de execução inicia com a pesquisa dos registros, através dos índices clusterizados, nas tabelas Vendas e Vendidos, que em seguida, tem a junção realizada com o método *hash match*. Com o resultado dessa junção e com os registros da tabela Produtos, também pesquisados através de um índice clusterizado, é realizada a segunda junção, desta vez aplicando o método *nested loops*. Finalizando, os registros são ordenados pela descrição do produto, devido a cláusula *distinct* presente nesse campo, e projetados.

6.6.4 SOLUÇÃO

A consulta reescrita na versão 3.0 do banco de dados sofre alterações na relação entre as tabelas, nomes dos campos e no parâmetro do código do cliente, antes representado pelo valor 62660 e que passa a ser representado pelo valor 29C4C89E-E5AC-4D4A-83C1-343F9991B654, que identifica a chave do registro na tabela Clientes.

Para otimizar a consulta foi criado o índice não clusterizado Idx_DataCliente que compreende os campos DataHora e Cliente_Id, na tabela Vendas, que compõem a cláusula *where* da consulta.

Com a inserção desse índice o plano de execução para a ser o apresentado na Figura 28. Os passos do plano de execução também se mantém os mesmos, com alteração somente nos tipos de índices utilizados e nos tipos de junções realizadas.

```
SELECT distinct (p.Descricao)
        FROM Vendas v
        INNER JOIN Vendidos vend on v.Id= vend.Venda_Id
        INNER JOIN Produtos p on vend.Produto_Id= p.Id
WHERE v.DataHora BETWEEN '01/01/2010' and '03/31/2010'
        and v.Cliente_Id='29C4C89E-E5AC-4D4A-83C1-343F9991B654'
```

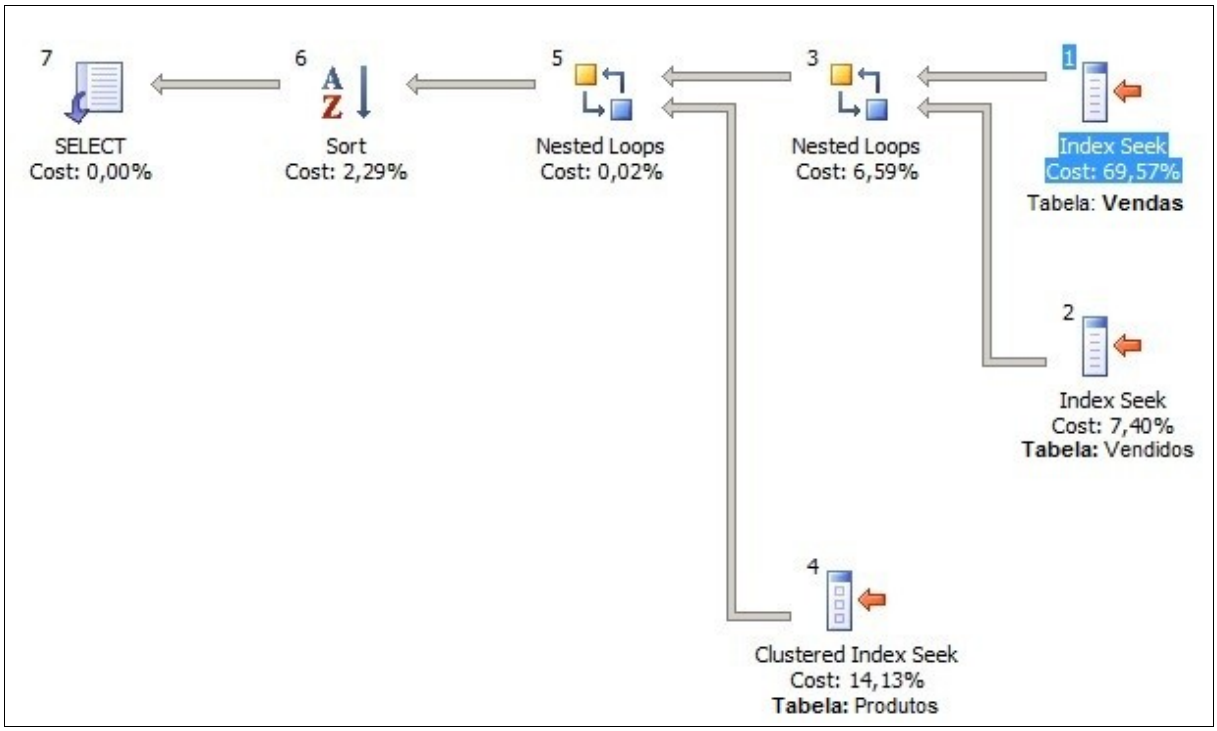


Figura 28: Plano de execução da consulta com *tuning* - Estudo de caso 5
Fonte: Autor

Ao realizar a pesquisa dos registros na tabela Vendas, primeiro passo do plano de execução, o índice criado Idx_DataCliente é utilizado produzindo assim uma busca com *index seek*. A seguir, também é utilizado um índice desse tipo para busca dos registros na tabela Vendidos, utilizando o índice Idx_VendaProduto criado no estudo de caso 2 na tabela Vendidos. Seguindo, é realizada a junção dessas duas tabelas, que juntamente com os registros da tabela Produtos, pesquisados com o auxilio do índice clusterizado da tabela, implementa a segunda junção. Finalizando, assim como na consulta original, os registros são ordenados pela descrição do produto e a projeção dos dados é realizada.

A Figura 29 apresenta os gráficos com os dados estatísticos da consulta original e da consulta com *tuning* e, a seguir, na Tabela 9, esses dados são tabulados.

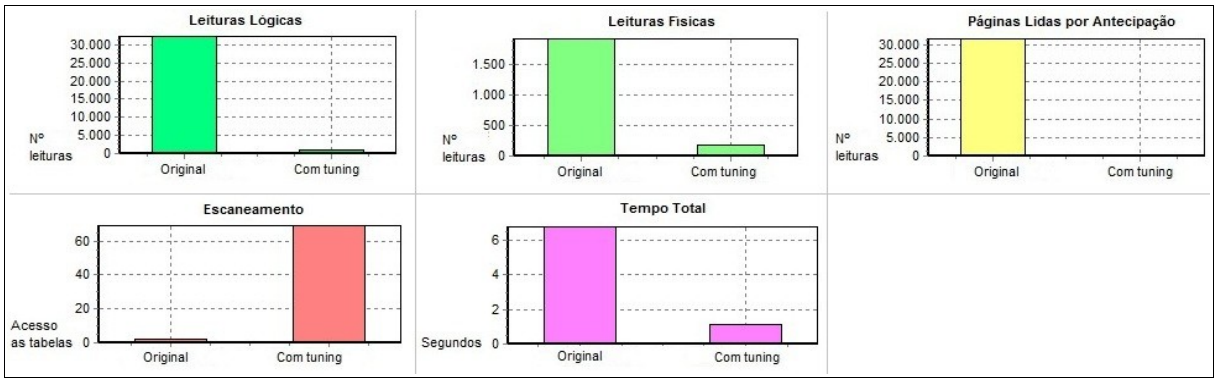


Figura 29: Gráficos comparativos - Estudo de caso 5
Fonte: Autor

Tabela 9: Dados estatísticos das consultas – Estudo de caso 5
Fonte: Autor

Informação Analisada	**SQL Original**	**SQL com *tuning***
Leituras lógicas	32322	854
Leituras físicas	1915	175
Páginas lidas por antecipação	31615	367
Escaneamento	2	69
Tempo total	7,56	1,16

Com a inserção do índice Idx_DataCliente na tabela Vendas a consulta apresentou uma melhora significativa em quatro dos cinco dados estatísticos analisados. O único que apresentou aumento na SQL com *tuning* foi a quantidade de escaneamentos realizados, mas isso se deve ao tipo de junção realizada para unir as tabelas Vendas e Vendidos. Na consulta original, a junção entre as tabelas Vendas e Vendidos é realizada utilizando *hash match* pois não existe índices nas colunas utilizadas na junção. Já na consulta com *tuning*, todos os campos utilizados para realizar a junção possuem índices relacionados, por isso o tipo de junção é *nested loop*.

Porém, nesse tipo de junção para cada registro da tabela interna é percorrida a tabela externa e feita a comparação segundo as condições de junção definidas (MSDN, 2009). Sendo assim, a tabela Vendas (tabela interna) é acessada uma única vez e a tabela Vendidos (tabela externa) é acessada 68 vezes, devido as condições impostas na cláusula *where*, para realizar a junção. Somando a quantidade de escaneamento realizados tem-se o valor de 69.

6.7 ESTUDO DE CASO 6 – *HINTS* DE JUNÇÃO

O sexto estudo de caso tem como objetivo demonstrar o uso dos *hints* de junção *hash join*, *loop join* e *merge join*. Tendo como base a consulta original, serão aplicados os três *hints* de junção e realizada as comparações entre os mesmos.

6.7.1 REGRA DE NEGÓCIO

A venda a prazo, apesar dos riscos provenientes, ainda é uma modalidade de venda bastante difundida, principalmente nas farmácias onde a relação com o cliente é de confiança. Mas, para que se mantenha o controle dessas vendas e, principalmente, do saldo devido por cada cliente, deve-se ter a possibilidade de realizar a consulta do saldo devedor de cada cliente.

6.7.2 PROBLEMA

A consulta, para retornar as informações necessárias para controle das contas a prazo em aberto, envolve no mínimo três tabelas, tendo assim que realizar duas junções. As tabelas envolvidas nessa consulta possuem uma grande quantidade de registros, o que interfere no desempenho das junções.

6.7.3 SQL ORIGINAL

A consulta original é composta por três tabelas: Vendas, que armazena o cabeçalho das vendas realizadas; Finaprazo, que armazena as parcelas das contas a prazo a receber e recebidas e, Clientes, que armazena o cadastro de clientes do estabelecimento. Para retornar apenas as contas em aberto em uma determinada empresa, deve-se filtrar, além da empresa, somente as que possuem o valor do saldo maior que zero.

A seguir é apresentada a consulta original e, na Figura 30, apresentado seu plano de execução.

```
SELECT f.Venda, c.Codigo, c.Nome, f.Parcela, f.Vencimento, f.Saldo
        FROM Vendas v
        INNER JOIN Finaprazo f on v.Venda= f.Venda and v.Empresa= f.Empresa
        INNER JOIN Clientes c on f.Codigo = c.Codigo
WHERE f.Saldo > 0 and v.Empresa=1
```

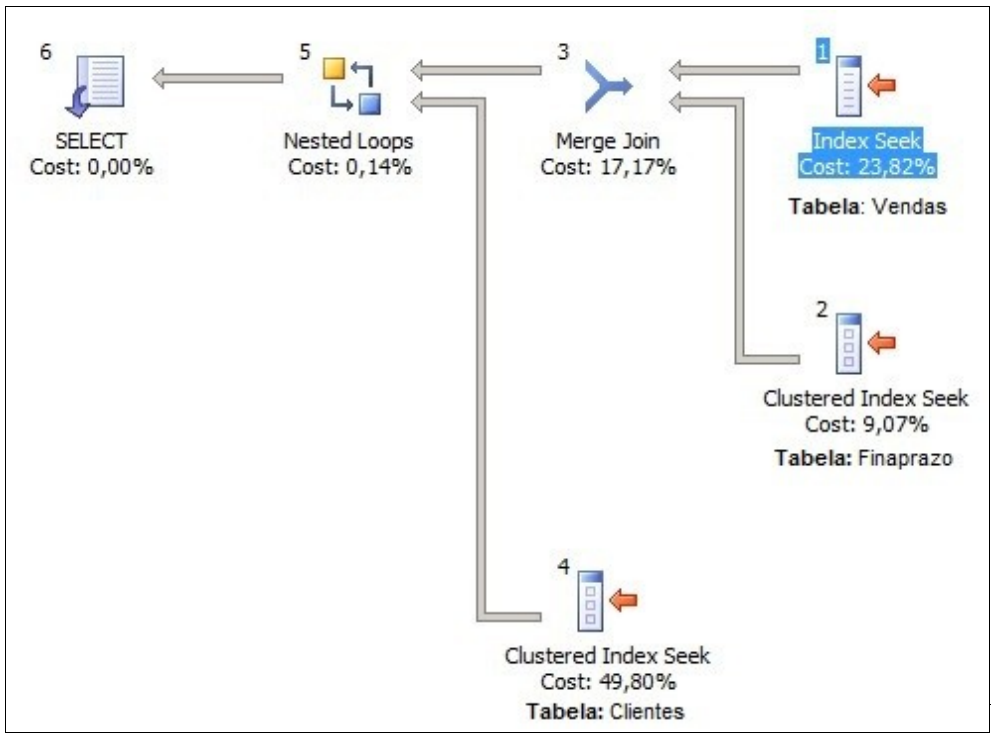


Figura 30: Plano de execução da consulta original - Estudo de caso 6
Fonte: Autor

O plano de execução inicia com a junção do tipo *merge join* entre as tabelas Vendas e Finaprazo, que tem seus registros pesquisados com o auxilio de índices não clusterizados e clusterizado, respectivamente. Após a operação de *merge join*, é realizada a segunda junção, dessa vez com a tabela Clientes, que resulta em uma junção do tipo *nested loops*. Finalizando, é realizada a seleção dos dados.

6.7.4 SOLUÇÃO

Com a reescrita da consulta com a versão 3.0 do banco de dados, a quantidade de tabelas envolvidas passa de três para quatro, além de envolver uma subconsulta na operação. Das tabelas envolvidas nessa consulta, a tabela Vendas equivale a tabela Venda da consulta

anterior, bem como a tabela VendasAPrazo equivale a Finaprazo e a tabela Pessoas equivale a Clientes. Além disso, para saber as contas que ainda não foram pagas deve-se pesquisar na tabela PagamentosVendasAPrazo. A equivalência do campo UnidadeNegocio_Id com o campo Empresa é a mesma apresentada no estudo de caso 1.

Sendo assim, o objetivo é avaliar com qual tipo de junção a consulta passa a ser executada com maior desempenho. Para isso, foram aplicados os *hints* de junção *hash join*, *loop join* e *merge join* na consulta abaixo e analisados os resultados encontrados. O modo de aplicação do *hint* foi inserindo a cláusula *option* (*hash join* | *loop join* | *merge join*) ao fim do comando SQL.

Na Figura 31, é apresentado o plano de execução da consulta original, sem aplicação de *hints* de junção.

```
SELECT v.Venda, p.Codigo,  p.Nome, vp.Parcela, vp.DataVencimento, vp.Valor
        FROM Vendas v
        INNER JOIN VendasAPrazo vp on v.Id=vp.Venda_Id
        INNER JOIN Pessoas p on v.Cliente_Id= p.Id
WHERE NOT EXISTS
        (SELECT pg.VendaAPrazo_Id
                FROM PagamentosVendasAPrazo pg
        WHERE pg.Parcela=vp.Parcela and pg.VendaAPrazo_Id=vp.Id)
and v.UnidadeNegocio_Id= '586B3FF2-1230-43C9-A557-114319E672EC'
```

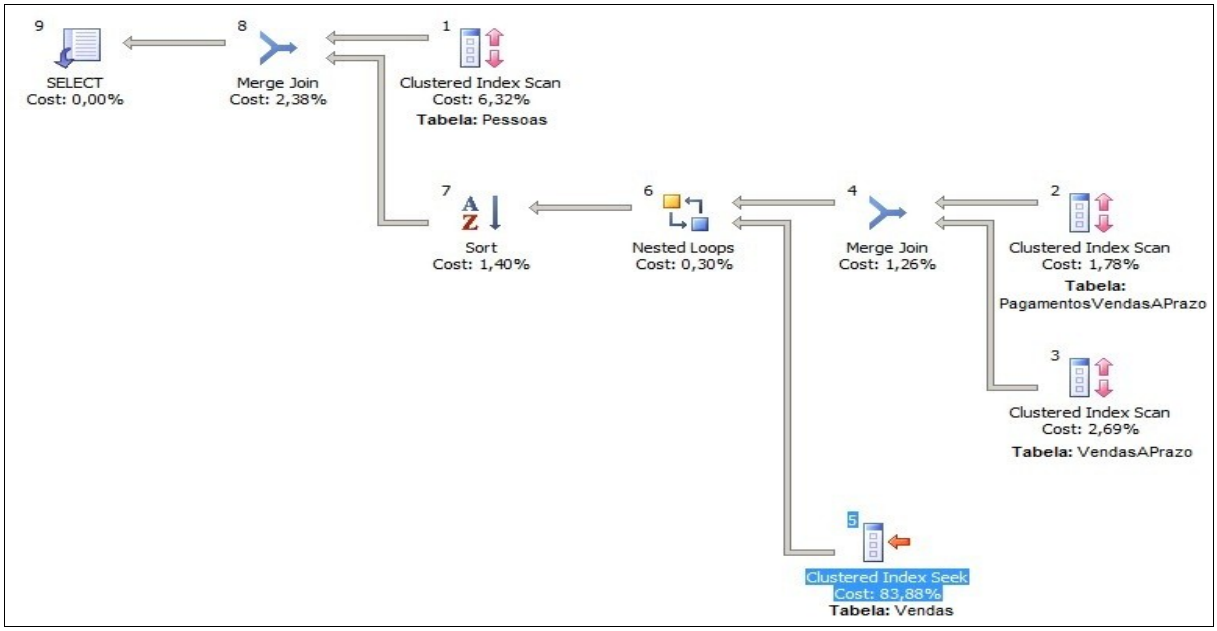


Figura 31: Plano de execução da consulta sem aplicação de *hints* de junção - Estudo de caso 6
Fonte: Autor

Como já mencionado acima, a consulta passa a envolver quatro tabelas. A primeira junção fica implícita na subconsulta, utilizando uma junção do tipo *merge* para retornar os pagamentos de contas a prazo já efetuados. Para isso, utiliza os índices das chaves primárias das tabelas PagamentosVendasAPrazo e VendaAPrazo. Com o resultado dessa junção, é realizada a segunda junção, inserindo para isso os dados da tabela Vendas que também são pesquisados utilizando o índice clusterizado dessa tabela. Seguindo, é realizada a ordenação dos registros retornados da segunda junção. Essa ordenação é realizada, pois o tipo de junção *merge*, que é a terceira junção executada, ordena as linhas relevantes previamente de acordo com o campo chave utilizado na junção, no caso o campo Cliente_Id. Para a terceira junção é utilizada também a tabela Pessoas. Finalizando, a seleção dos dados é executada.

Na Figura 32 são apresentados os gráficos comparando os dados estatísticos da consulta original, da consulta com aplicação do *hint hash join*, da consulta com aplicação do *hint loop join* e da consulta com aplicação do *hint merge join*, respectivamente. A Tabela 10 tabula esses resultados.

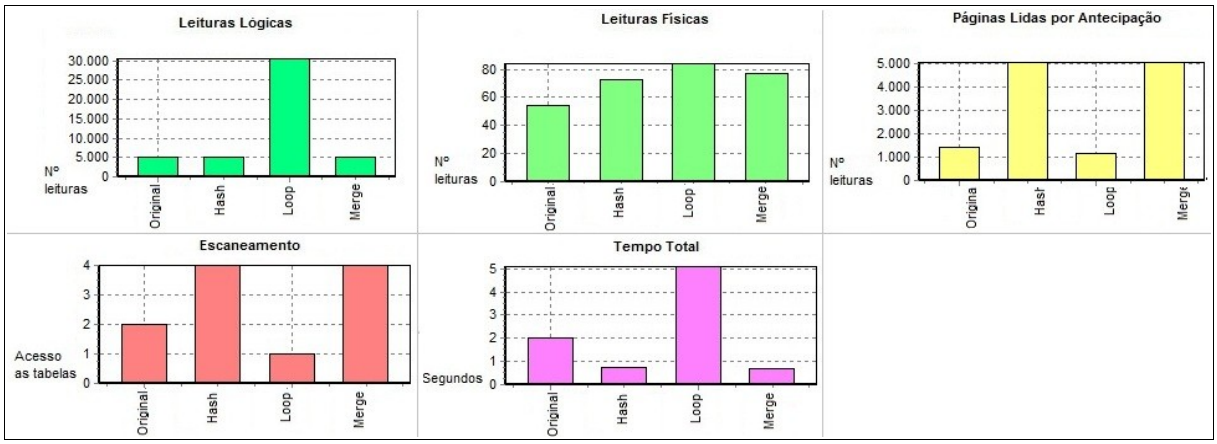


Figura 32: Gráficos comparativos - Estudo de caso 6
Fonte: Autor

Tabela 10: Dados estatísticos das consultas – Estudo de caso 6
Fonte: Autor

Informação Analisada	**SQL Original**	**SQL com *hint hash join***	**SQL com *hint loop join***	**SQL com *hint merge join***
Leituras lógicas	5099	5060	30509	5053
Leituras físicas	54	73	84	77
Páginas lidas por antecipação	1411	5033	1162	5033
Escaneamento	2	4	1	4
Tempo total	1,99	0,73	5,09	0,67

Através dos resultados gerados pode-se comprovar, que para a consulta analisada, o tipo de junção mais eficiente é a junção utilizando *merge join*, seguindo pela junção do tipo *hash join*. A junção com *loop join* foi a que apresentou o pior resultado.

O *hint merge join* é indicado quando as tabelas envolvidas são de médio a grande porte[8], além da quantidade de linhas retornadas ser grande (MSDN, 2009; SACK, 2005). Essas condições são aplicadas as tabelas envolvidas na junção, por isso esse *hint* apresentou o melhor resultado.

O *hint hash join* também apresentou resultados próximos ao *hint merge join*. E isso deve-se ao fato desse *hint* ser recomendado para uso em junções onde as tabelas envolvidas são grandes, porém não possuem índices nas colunas utilizadas na junção (MSDN, 2009; SACK, 2005). Essas duas condições são contempladas nas tabelas e campos utilizados para a junção.

Já o *hint loop join*, que apresentou o pior resultado tendo seus dados estatísticos superiores até mesmo a consulta original, é recomendado a ser utilizado em junções onde uma das tabelas é pequena e a outra grande (MSDN, 2009; SACK, 2005). Como esse não era o caso de nenhuma das tabelas envolvidas, o tempo de total de execução superou mais de sete vezes o melhor resultado obtido. Além disso, conforme já mencionado na seção 3.3.3, esse tipo de junção, para cada linha da tabela interna, percorre a tabela externa e realiza as comparações segundo as condições de junção definidas (MSDN, 2009). Sendo assim, gera um produto cartesiano ao percorrer todas as linhas, gerando todas as combinações possíveis. Esse processo tem um alto custo que interfere diretamente no desempenho da consulta, conforme comprovado pelo estudo de caso realizado.

6.8 ESTUDO DE CASO 7 – *HINT* DE TABELA *INDEX*

O último estudo de caso tem como objetivo demonstrar o uso do *hint* de tabela *index*, que força o otimizador de consulta a utilizar um índice definido.

8 Não há definição, pelos autores citados no parágrafo, do método a ser considerado para classificar uma tabela quanto a seu porte. Por isso, levou-se em consideração a quantidade de registros da maior tabela do sistema estudado para dimensionar o porte das demais tabelas.

6.8.1 REGRA DE NEGÓCIO

Vários indicadores precisam ser gerenciados constantemente em uma farmácia e dentre eles esta o *ticket* médio, que indica o valor médio das vendas realizadas. Geralmente as estratégias de venda tem como objetivo o aumento do valor do *ticket* médio, por isso deve ser possível recuperar esse importante parâmetro do sistema de gerenciamento da farmácia.

6.8.2 PROBLEMA

O grande problema enfrentado nesse tipo de consulta é a quantidade de registros que precisam ser analisados para a aplicação da função que calcula a média dos valores. Na base de dados analisada, são mais de 391.000 registros. Além disso, com a possibilidade de escolha de mais de uma unidade de negócio para a obtenção do *ticket* médio, é necessário utilizar os predicados *in* ou *or* como critérios de restrição da consulta. Porém, conforme já citado no capítulo 3, o uso desses predicados interferem no uso de índices nas consultas.

6.8.3 SQL ORIGINAL

O valor do *ticket* médio pode ser extraído consultando somente a tabela Caixa restringindo apenas os registros de vendas, conforme já demonstrado no estudo de caso 1, somando os valores das diferentes formas de pagamento da venda e após, aplicada a função interna que calcula a média dos valores encontrados. Na tabela Caixa há índices não clusterizados para os três atributos da cláusula *where* da consulta abaixo.

```
SELECT
avg(c.Dinheiro + c.Cheque + c.Cartao + c.Aprazo +c.Convenio +  c.Outros + c.Cheque_pre) as total
        FROM Caixa c
WHERE c.Data between '01/01/2009' and '01/01/2010' and c.Empresa in (1,2)
        and c.Lancamen in ('CH','CP','CR','VC','VP','VV')
```

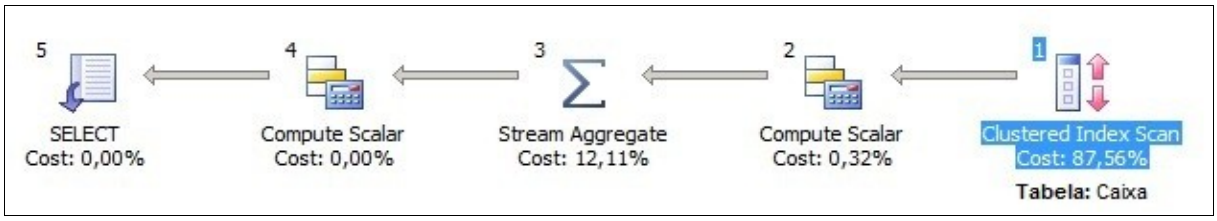


Figura 33: Plano de execução da consulta original - Estudo de Caso 7
Fonte: Autor

O plano de execução, apresentado na Figura 33, inicia realizando a busca pelos registros que correspondem a cláusula *where* através do índice da chave primária da tabela Caixa. Após, a soma dos campos destacados é realizada e aplicada a função que retorna a média dos valores previamente somados. Finalizando, é projetado o resultado com o valor do *ticket* médio calculado.

6.9 SOLUÇÃO

Com a alteração na estrutura do banco de dados, se a consulta na tabela Caixas envolve restrição de data e unidade de negócio, deve-se realizar a junção com a tabela Vendas, pois somente essa possui o campo que identifica a unidade de negócio e a data. Sendo assim, a consulta reescrita passa a utilizar duas tabelas. Os campos Empresa, da consulta original, e o campo UnidadeNegocio_Id são equivalentes, conforme já explicado no estudo de caso 1.

O campo UnidadeNegocio_Id, que utiliza o predicado *in* para a realização da busca, não possui um índice composto apenas por esse campo. Porém, mesmo se esse índice existisse não seria utilizado na consulta, pois a utilização do predicado *in* na cláusula *where* impede o uso de índices. Sendo assim, para forçar o uso de índices foi aplicado o *hint index*. Para a solução foram aplicados dois *hints*, sendo que o primeiro é aplicado na tabela Caixas onde é forçado o uso do índice Idx_FormaPagamento e o segundo é aplicado na tabela Vendas, utilizando o índice Idx_DataUnidadeNegocio para realizar a busca pelos registros. Ambos os índices utilizados foram criados no primeiro estudo de caso apresentado.

A seguir, é apresentada a consulta reescrita com utilização do *hint index*. A Figura 34, representa o plano de execução dessa consulta, se a mesma fosse executada sem a utilização do *hint*. Já a Figura 35, apresenta essa mesma consulta com aplicação do *hint*.

```
SELECT  avg(c.ValorTotal) as total
        FROM Caixas c  WITH(INDEX(Idx_FormaPagamento) )
        INNER JOIN Vendas v WITH(INDEX(Idx_DataUnidadeNegocio) ) on v.Id=c.Venda_Id
WHERE  v.DataHora between '01/01/2009' and '01/01/2010' and
          v.UnidadeNegocio_Id IN ('586B3FF2-1230-43C9-A557-114319E672EC',
           'B803D5C1-5CE5-489E-A7BE-B7023E7421C7')
```

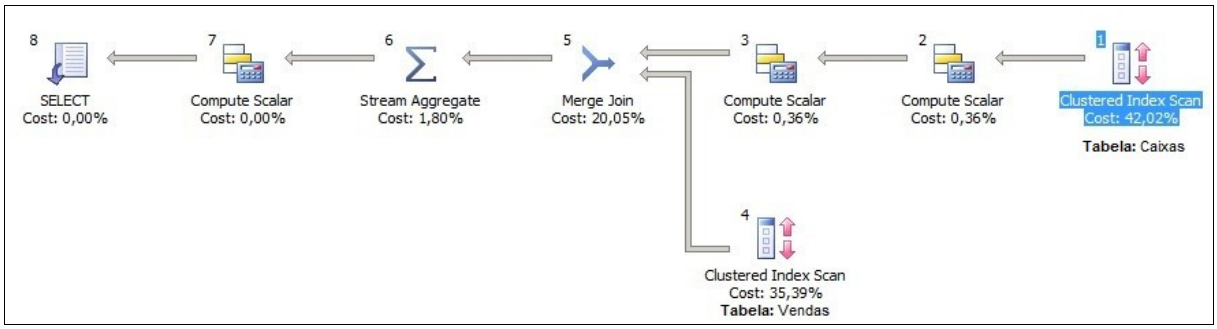


Figura 34: Plano de execução da consulta sem *hint* - Estudo de caso 7
Fonte:Autor

No primeiro passo do plano de execução apresentado acima, é realizada a busca dos registros da tabela Caixas utilizando o índice clusterizado da mesma. Após, é realizada a soma dos valores que contemplam o campo computado ValorTotal. Seguindo, no quarto passo é realizada a busca dos registros que contemplam a cláusula *where* na tabela Vendas, também utilizando o índice clusterizado dela. Após, a junção das tabelas é executada através do método *merge join* e, finalizando é aplicada a função *avg* e retornado o registro.

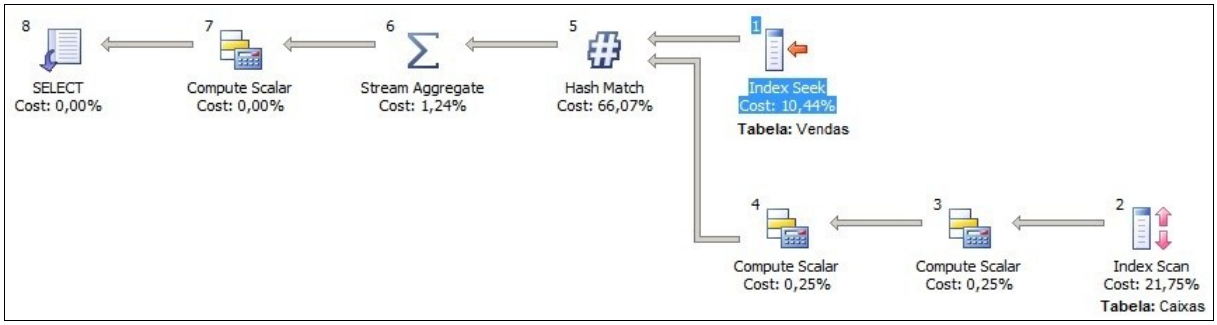


Figura 35: Plano de execução da consulta com *hint* - Estudo de caso 7
Fonte: Autor

No primeiro passo do plano de execução é realizada a busca dos registros conforme a restrição da cláusula *where*, que para isso utiliza o índice Idx_DataUnidadeNegocio. Após, é

realizada a soma dos valores que contemplam o campo computado ValorTotal. A seguir, o índice Idx_FormaPagamento é utilizado para seleção dos registros da tabela Caixas. Após é realizada a junção através do *hash match* entre as tabelas Caixas e Vendas e, finalizando, é aplicada a função *avg* e retornado o registro.

Como pode ser visualizado nos gráficos apresentados na Figura 36 e na Tabela 11, há uma queda significativa entre a consulta original e a consulta com uso de *hint*.

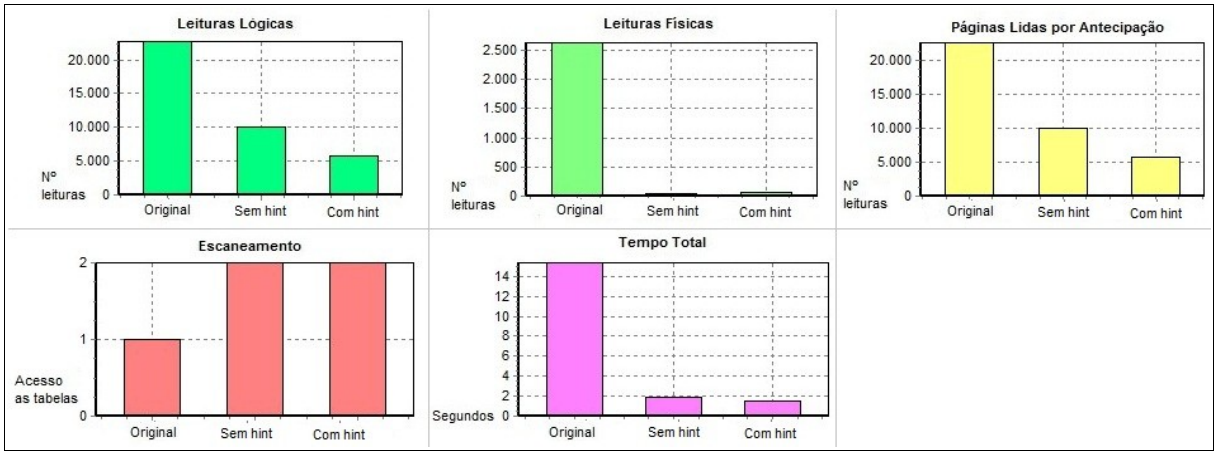


Figura 36: Gráficos comparativos - Estudo de caso 7
Fonte: Autor

Tabela 11: Dados estatísticos das consultas - Estudo de caso 7
Fonte: Autor

Informação Analisada	**SQL Original**	**SQL sem *hint***	**SQL com *hint***
Leituras lógicas	22700	10105	5773
Leituras físicas	2627	33	54
Páginas lidas por antecipação	22564	10066	5766
Escaneamento	1	2	2
Tempo total	15,37	1,89	1,49

No estudo de caso apresentado, o uso do *hint* de tabela *index* apresentou um bom resultado, reduzindo em mais de dez vezes o tempo de execução da consulta, além da redução significativa de leituras lógicas e físicas realizadas.

Porém, a consulta executada sem o *hint* de tabela *index* também apresentou um bom resultado. Sendo assim, a escolha pelo uso de *hints* deve ser criteriosa para que o resultado

seja o esperado. O uso incorreto de um *hint* pode prejudicar a consulta ao invés melhorá-la. Por isso, recomenda-se revisar o uso e a necessidade do *hint* na consulta, pois se as circunstâncias mudarem o *hint* pode não ser mais necessário.

6.10 CONSIDERAÇÕES FINAIS

Esse capítulo apresentou a aplicação das técnicas de *tuning* estudadas no Capítulo 3 em sete estudos de casos. A seguir, a Tabela 12 apresenta as considerações para aplicação de cada uma das técnicas aplicadas nos estudos de caso.

Tabela 12: Principais considerações sobre as técnicas de *tuning* aplicadas
Fonte: Autor

Técnica de *tuning*	**Considerações**
Agrupamento	A criação de índices com os campos que fazem parte do agrupamento e das agregações auxiliam na otimização das consultas com o predicado *group by*.
Ordenação	Para otimizar o uso da ordenação, é importante conhecer em que casos o SQL *Server* utiliza os métodos de ordenação internamente para que essa operação não se torne redundante.
Uso efetivo de índices – predicado de igualdade	O predicado de igualdade está dentre os mais utilizados nas consultas, por isso a otimização de seu uso é importante. Porém, quando criado índices deve-se avaliar quais campos farão parte desse índice, para que o desempenho da consulta seja o melhor possível.
Uso efetivo de índices – predicado *like*	O predicado *like* deve ser estruturado de forma a utilizar o(s) índice(s) que contemplam os campos envolvidos nessa operação. Conforme Tabela 1, é utilizado o índice de forma efetiva quando esse predicado é apresentado na forma *like* <literal>%
Uso efetivo de índices – predicado *between*	Assim como no predicado de igualdade, a criação de índices em consultas com o predicado *between* deve passar por avaliação dos campos que farão parte do índice.
Hints de junção	O uso de *hints*, de qualquer tipo, deve ser sempre criterioso e executado em último caso, pois seu uso requer revisão periódica das consultas que o utilizam. Se bem aplicado apresenta bons resultados, caso

Técnica de *tuning*	**Considerações**
	contrário pode piorar o desempenho da consulta.
Hint de tabela - *index*	Geralmente o otimizador do SQL *Server* escolhe corretamente o índice a ser utilizado, porém é possível forçar o uso de um índice quando, por exemplo, é utilizado um predicado do tipo *nonsargable*.

Nem sempre a aplicação de técnicas de *tuning* e estruturação da SQL, utilizando as melhores práticas, resolvem o problema de desempenho de uma consulta, pois há casos que fogem às regras ou técnicas previamente definidas. Sendo assim, o *tuning* de consulta não deve ser aplicado de forma automática e sem critérios.

Com os estudos de casos realizados foi possível identificar que a utilização das técnicas de *tuning*, se bem aplicadas, pode otimizar as consultas de forma satisfatória. Às vezes, a redução do tempo de execução pode parecer pequena, mas levando em consideração a quantidade de execuções da consulta diariamente essa pequena redução torna-se significativa.

Foi considerado como critério para cada consulta analisada os dados com maior número de registros, ou seja, os sete estudos de caso foram analisados para os piores casos de cada consulta.

Para a criação dos índices, nos estudos de caso onde houve a necessidade destes para melhorar o desempenho das consultas, foram considerados os campos que melhor apresentaram resultados nos dados estatísticos analisados. Além disso, foi levado em consideração o estudo sobre os predicados apresentados no Capítulo 3.

Durante o período de execução do projeto de *tuning* de consulta, os estudos de caso foram reanalisados com frequência, a fim de avaliar se as alterações realizadas para os estudos de caso não interferiram nos resultados de outros estudos de caso. Ao final da execução do projeto de *tuning* de consulta, foi realizada uma reanalise total dos sete estudos de caso.

7 CONCLUSÃO

A aplicação de *tuning* pode ser realizada nas diversas áreas de um sistema, iniciando desde a sua concepção, com a descrição das regras de negócio, passando pelos projetos de banco de dados e estrutura lógica e física do sistema. Sendo assim, a aplicação de *tuning* de consulta, apresentada nesse trabalho, é apenas uma pequena amostra das melhorias de desempenho que podem ser alcançadas em um sistema.

A aplicação de *tuning* não é uma tarefa simples, exige conhecimento e monitoramento constante da aplicação como um todo. Por isso, antes de iniciar qualquer alteração, é necessário dedicar tempo ao estudo e interpretação das técnicas de *tuning*, pois cada consulta é única e nem sempre as orientações sobre as técnicas são completamente aplicáveis ao problema de desempenho enfrentado. Às vezes, a redução do tempo de execução pode parecer pequena, mas levando em consideração a quantidade de execuções da consulta diariamente essa pequena redução torna-se significativa.

Existem várias técnicas de *tuning* que podem ser aplicadas, sendo que a maioria delas envolve alteração da estrutura da SQL. Sendo assim, durante as fases iniciais de desenvolvimento de um *software* deve-se ter como premissa criar um sistema que apresente desempenho satisfatório. Para se alcançar esse objetivo, dois pontos que merecem atenção são o projeto de banco de dados, que deve ser criado de forma a otimizar as futuras consultas realizadas, e a escrita das SQLs que deve ser realizada de forma a aplicar as melhores práticas de estruturação de consultas.

A fim de alcançar esses dois objetivos, foram criados os projetos de banco de dados e o projeto de *tuning* de consulta, que tiveram como base as tabelas envolvidas na funcionalidade de frente de caixa, disponível no sistema de gerenciamento escolhido.

A execução do projeto de banco de dados contou com duas atividades principais. A primeira delas foi a criação do modelo relacional das tabelas envolvidas nas operações de frente de caixa, que levou em consideração, além das regras de negócio atuais, novas funcionalidades e também a aplicação de conceitos de sistemas de bancos de dados. A segunda atividade, foi a implementação de uma ferramenta para importação dos dados do banco de dados de versão 2.0 para a versão 3.0. O principal desafio nessa fase foi manter os dados íntegros, já que as estruturas das tabelas mudaram completamente.

Com a importação dos dados realizada, foi iniciado o projeto de *tuning* de consulta, aplicando as técnicas de *tuning,* destacadas na seção 4.3, em sete estudos de caso que destacam algumas consultas críticas da funcionalidade de frente de caixa. O maior desafio enfrentado na aplicação de *tuning* de consulta, foi identificar qual técnica apresentaria melhores resultados para a consulta que estava sendo analisada, pois em uma mesma consulta havia a possibilidade de aplicação de diferentes técnicas. Com os estudos de caso realizados, pode-se comprovar que a reescrita das consultas juntamente com a reestruturação do banco de dados e criação de índices, melhora o desempenho das consultas. Um banco de dados estruturado com chaves primárias e relacionamentos identificados, auxilia na busca pelos registros e, consequentemente, diminui o tempo de execução da consulta.

Para projetos futuros, pode-se avaliar a aplicação de *tuning* não só em *software*, mas também em *hardware*, analisando como esse tipo de *tuning* interfere no desempenho de uma aplicação, pois dependendo do problema de desempenho que o sistema se encontra, apenas a aplicação de *tuning* de consulta pode não apresentar resultados satisfatórios. Sendo assim, o *tuning* em *hardware* ganha importância, procurando deixar que *software* e *hardware* trabalhem em conjunto e de forma equilibrada para que seja possível extrair o máximo de desempenho dessa combinação. Outra sugestão de projeto futuro, seria a comparação de aplicação de *tuning* de consulta do SQL *Server* com outro SGDB.

8 REFERÊNCIAS

BALTER, Alison. ***Microsoft SQL Server 2005 Express in 24 hours***. USA: Sams, 2006.

BEVAART, Nils. **SQL *Server Performance*: *Query Tuning vs. Process Tuning***. In: SQL Server Performance.com. 2006. Disponível em: <http://www.sql-server-performance.com/articles/per/query_process_tuning_p1.aspx>. Acessado em 02 de maio de 2010

BRYLA, Bob; LONEY, Kevin. ***Oracle Database* 11g:** Manual do DBA. Porto Alegre: Bookman, 2009.

CHAN, Immanuel. ***Oracle® Database Performance Tuning Guide* 10*g* *Release* 2 (10.2)** . 2008. Disponível em: <http://download.oracle.com/docs/cd/B19306_01/server.102/b14211.pdf>. Acessado em 14 de abril de 2010.

DEWSON, Robin. ***Microsoft SQL Server* 2008 para Desenvolvedores:** De Iniciantes ao Profissional. Rio de Janeiro: Alta Books, 2008.

ELMASRI, Ramez; NAVATHE, Shamkant B.. **Sistemas de Banco de Dados**. 4.ed. São Paulo: Addison-Wesley, 2005.

FRITCHEY, Grant; DAM, Sajal. ***SQL Server 2008 Query Performance Tuning Destilled***. New York: Springer, 2009.

GULUTZAN, Peter; PELZER, Trudy. **SQL *Performance Tuning*.** Boston: Pearson, 2003.

IBM. **Ajustando a Capacidade e as Definições de Hardware.** 2004. Disponível em: <http://publib.boulder.ibm.com/infocenter/wasinfo/v5r1//index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/tprf_tunehdwcap.html> . Acessado em 27 de

abril de 2010

MCGEHEE, Brad. ***Beware: New Query Hints Added to SQL Server 2005***. In: SQL Server Performance.com. 2007. Disponível em: <http://www.sql-server-performance.com/articles/per/new_query_hints_p1.aspx>. Acessado em 23 de maio de 2010.

MCGEHEE, Brad. ***Performance Tuning* SQL *Server Joins***. In: SQL Server Performance.com. 2006a. Disponível em:<http://www.sql-server-performance.com/tips/tuning_joins_p1.aspx>. Acessado em 11 de maio de 2010

MCGEHEE, Brad. ***SQL Server Optimizer Hint***. In: SQL Server Performance.com. 2006b. Disponível em: <http://www.sql-server-performane.com/tips/hints_general_p1.aspx>. Acessado em 22 de maio de 2010.

MICROSOFT. **Comparação de Funcionalidades do SQL *Server* 2005**. 2005. Disponível em: <http://www.microsoft.com/portugal/sql/prodinfo/features/compare-features.mspx>. Acessado em 07 de junho de 2010

MICROSOFT. ***Microsoft* SQL *Server* 2008**. 2010. Disponível em: <http://www.microsoft.com/sqlserver/2008/pt/br/default.aspx>. Acessado em 16 de abril de 2010

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

MSDN. **Manuais *Online:* SQL *Server* 2008**. 2009. Disponível em: <http://msdn.microsoft.com/pt-br/library/ms130214%28v=sql.100%29.aspx>. Acessado em 01 de maio de 2010.

ORACLE. ***Oracle 8i Designing and Tuning for Performance***, Release 2 (8.1.6). 2000. Disponível em : <http://download.oracle.com/docs/cd/A87861_01/NT817EE/server.817/a76992/ch2_meth.htm>. Acessado em 06 de abril de 2010.

PILECKI, Maciej. **Como otimizar o desempenho da consulta no SQL *Server.*** In: Tecnet Magazine. Edição: Novembro/2007. Disponível em: <http://technet.microsoft.com/pt-br/magazine/2007.11.sqlquery.aspx>. Acessado em 08 de maio de 2010.

PRICE, Jason. ***Oracle Databse 11 g SQL***. Porto Alegre: Bookman, 2009.

POWELL, Gavin; ***Oracle Performance Tuning for* 10gR2**. 2ª ed. Massachusetts: Elsevier, 2007.

RAMALHO, José Antônio Alves. ***Microsoft* SQL *Server* 2005: guia prático**. Rio de Janeiro: Elsevier, 2005.

RIBEIRO, Paulo. ***Tuning*: Plano de Execução. In: SQL *Magazine*. 2004. Disponível em: <http://www.sqlmagazine.com.br/Colunistas/PauloRibeiro/09_Tunning_ExecutionPlan_PT2.asp>. Acessado em 08 de maio de 2010.

SACK, Josep. ***SQL Server 2005 T-SQL Recipes:*** *A Problem- Solution Approach*. New York: Apress, 2005.

SCHNEIDER, Rober D.; GIBSON, Darril. ***Microsoft SQL Server 2008 for Dummies.*** New Jersey: Wiley, 2008.

SHASHA, ***Dennis. Database Tuning****: A Principled Approach*. New Jersey: Prentice Hall, 1992.

SHASHA, Dennis; BONNET, Philippe. ***Database Tuning***. San Francisco: Morgan Kaufmann, 2003.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistemas de banco de dados**. 3.ed. São Paulo: Makrom Books, 1999.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. ***Database System Concepts***. 5.ed. New York: McGraw-Hill, 2005.

VAIDYANATHA , Gaja K.; **Oracle Database Performance Management**. 2001. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.4513&rep=rep1&type=pdf>. Acessado em: 02 de abril de 2010

ANEXO A

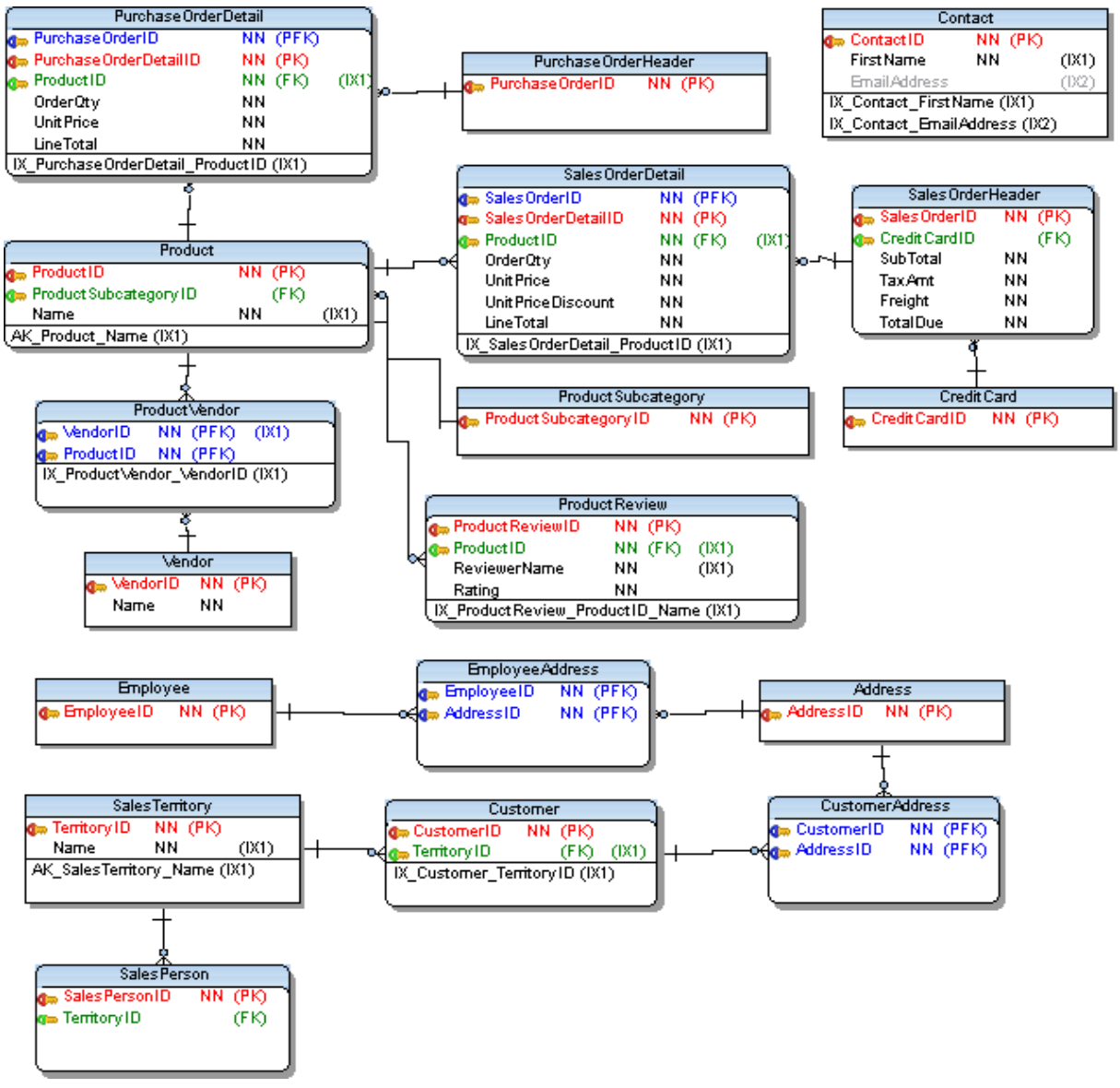
Ano	Versão	Descrição
1988	SQL *Server*	A aplicação SQL *Server* é criada junto com *Sybase* para uso na OS/2
1993	SQL *Server* 4.2	Um banco de dados de estação de trabalho, de baixa funcionalidade, capaz de atender ás necessidades de armazenamento de dados de um pequeno departamento. Foi integrado com o *Windows*, com uma interface fácil de usar.
1994		A *Microsoft* se separa da *Sybase*.
1995	SQL *Server* 6.05	A maior parte do mecanismo de banco de dados foi reescrita, gerando a primeira versão "significativa". O desempenho melhorou sendo possível manipular pequenas aplicações de *e-commerce* e de *intranet*. Seu valor custava uma fração do preço dos concorrentes.
1998	SQL *Server* 7.0	Mais uma vez uma parte do núcleo do mecanismo de banco de dados foi significativamente reescrito. Uma versão de definição, que oferecia um banco de dados razoavelmente poderoso e com muitos recursos, sendo um alterativa viável para os negócios de pequeno e médio porte. Conquistou uma boa reputação por sua facilidade de uso e por oferecer ferramentas fundamentais de negócio em seu pacote, sendo que nos concorrentes era um recurso adicional.
2000	SQL *Server* 2000	Com uma vasta melhora no desempenho, na escalabilidade e na confiabilidade, se tornou um grande *player* no mercado de banco de dados. Um grande aumento no preço não foi bem compreendido, embora ainda seja em torno da metade do preço do Oracle, mas a excelente gama de ferramentas de gerenciamento, desenvolvimento e análise conquistou novos clientes.
2005	SQL *Server* 2005	Muitas áreas foram reescritas, mas o maior salto é a introdução do f*ramework .NET*. Ele permitirá que os objetos específicos do SQL *Server .NET* sejam criados, dando ao SQL *Server* a funcionalidade flexível que o Oracle possui com a inclusão do Java.
2008	SQL *Server* 2008	O principal objetivo dessa versão é lidar com os muitos formatos que os dados disponíveis atualmente. Foi ampliada a infraestrutura da versão anterior oferecendo novos tipos de dados e o uso da Consulta Integrada de Linguagem (LINQ). Também lida com dados, como XML, dispositivos compactos e massivas de instalações de banco de dados que residem localidades distintas. Também oferece a habilidade de configurar regras dentro de um *framework* para garantir que banco de dados e objetos se adaptem aos critérios definidos.

ANEXO B

Modelo relacional utilizado para aplicação das técnicas de *tuning* do capítulo 3. A Tabela 13 apresenta a legenda dos campos dos modelos relacionais apresentados a seguir.

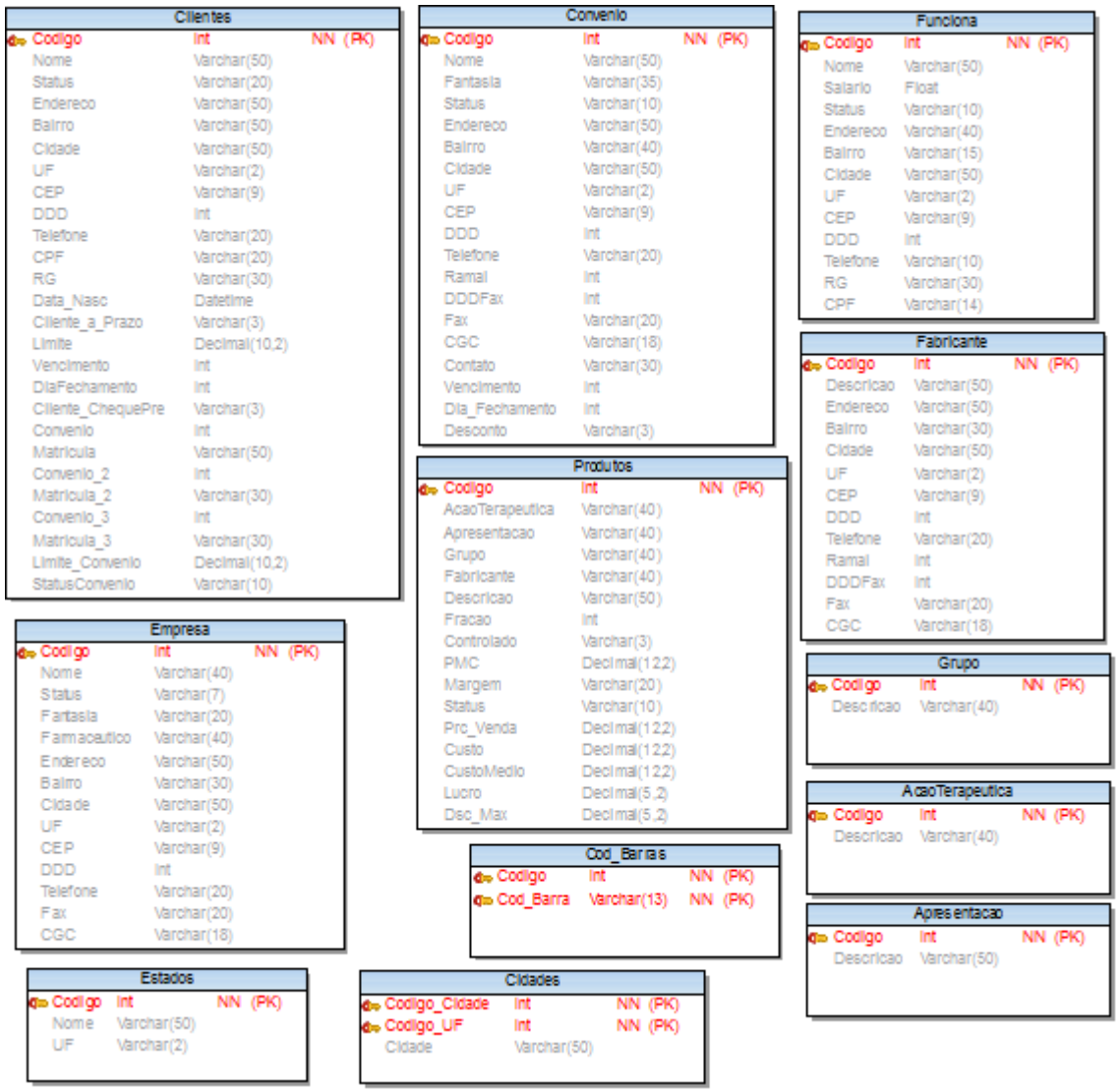
Tabela 13: Legenda de campos do modelo relacional

	Chave primária		Chave estrangeira		Chave estrangeria e primária
	Campo obrigatório		Campo não obrigatório		



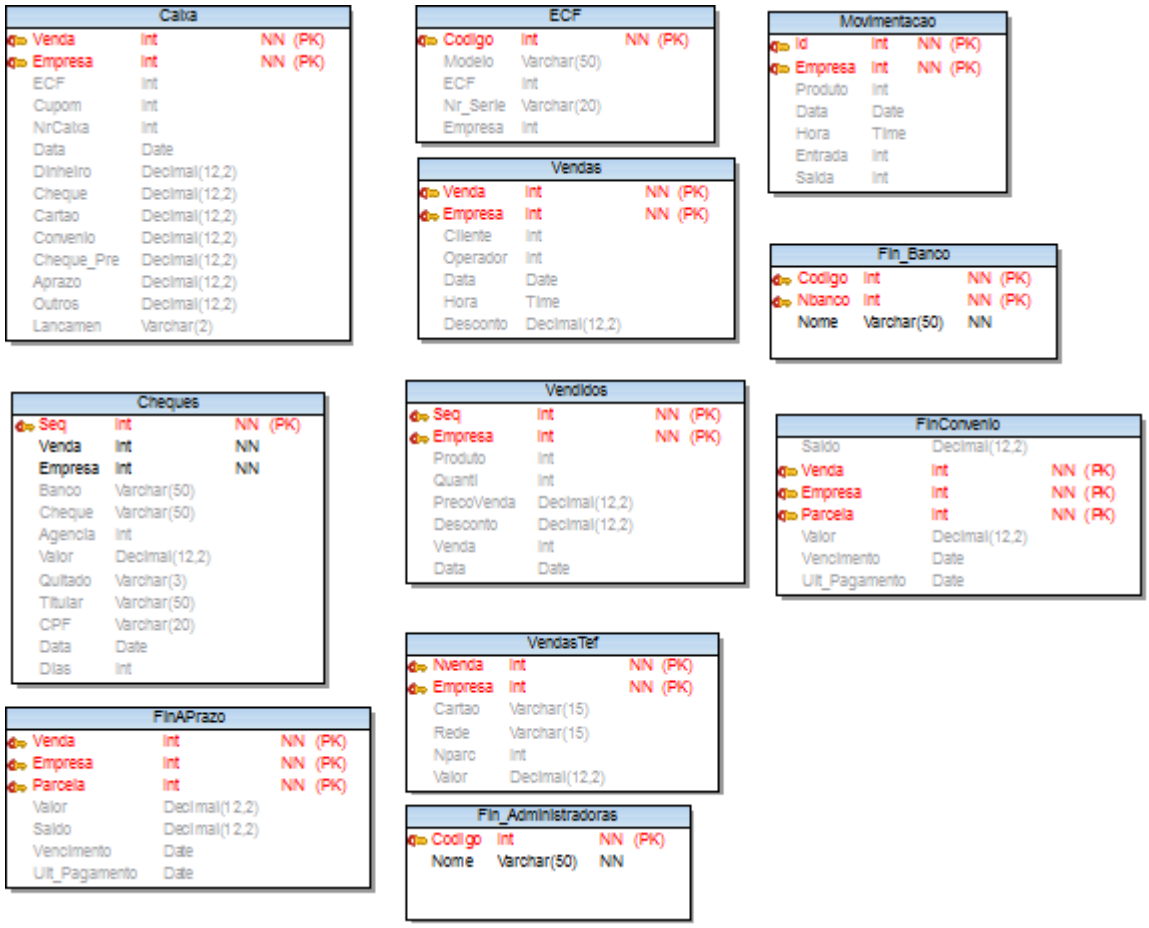
ANEXO C

Modelo relacional da versão 2.0 do banco de dados, que representa as tabelas relacionadas aos cadastros.



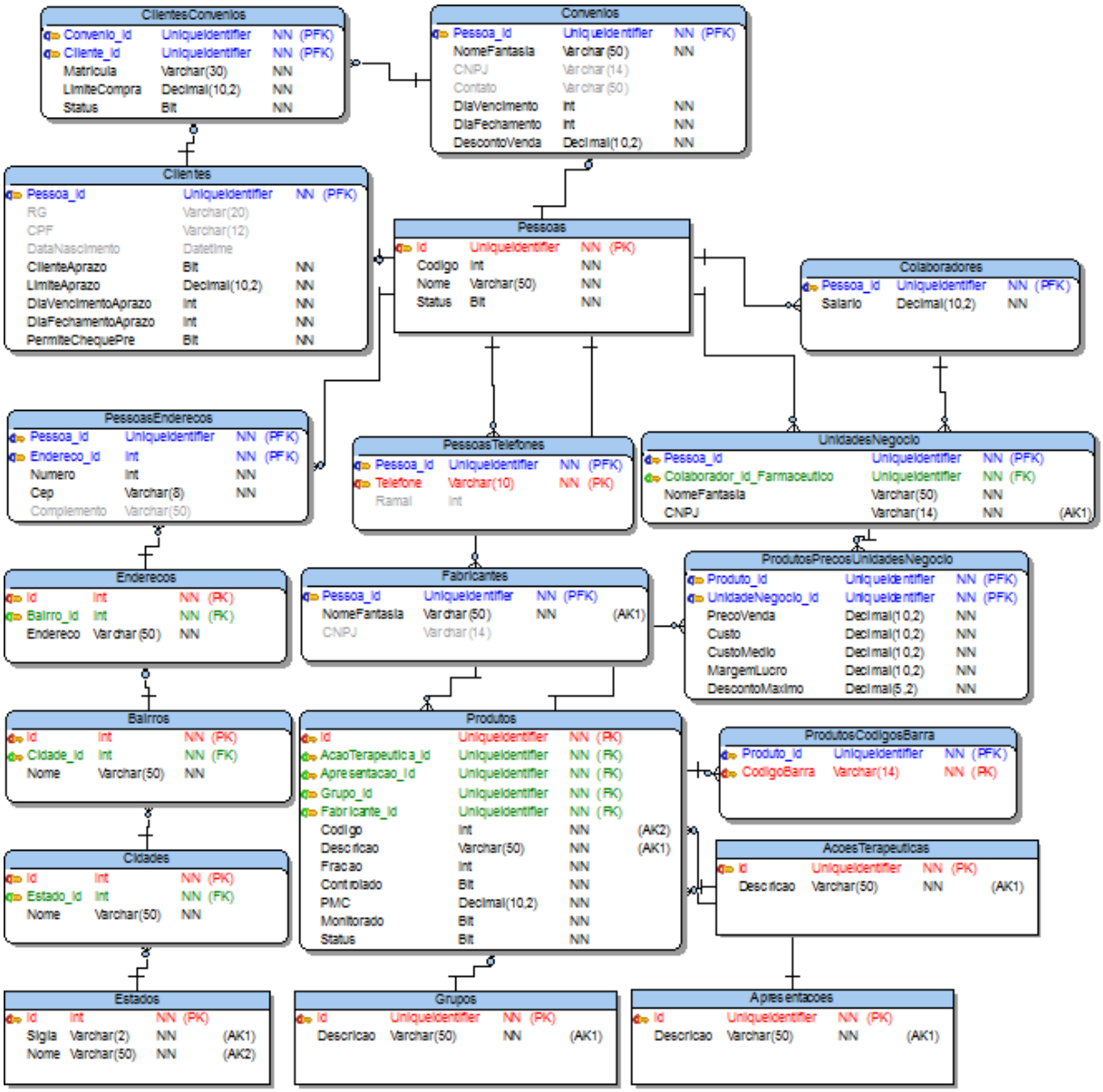
ANEXO D

Modelo relacional da versão 2.0 do banco de dados, que representa as tabelas relacionadas aos registros de vendas e pagamentos de contas.



ANEXO E

Modelo relacional da versão 3.0 do banco de dados, que representa as tabelas relacionadas aos cadastros.



ANEXO F

Modelo relacional da versão 3.0 do banco de dados, que representa as tabelas relacionadas aos registros de vendas e pagamentos de contas.