UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VIVIANNE CRISTINE ALVES

OTIMIZAÇÃO EM CONSULTAS NO SQL SERVER: ESTUDO DE CASO

JOÃO PESSOA
2024

VIVIANNE CRISTINE ALVES

OTIMIZAÇÃO EM CONSULTAS NO SQL SERVER: ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Ciência da Computação da Universidade Federal da Paraíba como requisito complementar para obtenção do título de Bacharel em Ciência da Computação, sob orientação do professor Sr. Derzu Omaia.

JOÃO PESSOA

2024

Catalogação na publicação
Seção de Catalogação e Classificação

A474o Alves, Vivianne Cristine.
Otimização de consultas no SQL server: estudo de caso / Vivianne Cristine Alves. - João Pessoa, 2024.
53 f. : il.

Orientação: Derzu Omaia.
TCC (Graduação) - UFPB/CI.

1. Banco de dados relacional. 2. Otimização. 3. SQL server. 4. Tuning. I. Omaia, Derzu. II. Título.

UFPB/CI CDU 004.6

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

À Vivi de alguns anos atrás.

AGRADECIMENTOS

Gostaria de agradecer primeiramente a minha família, minha tia Marlene, que durante metade da graduação me apoiava, minha tia Eliene, que me deu um notebook para qual eu pudesse estudar e minha mãe Edith que mesmo longe me incentivou.

Aos meus amigos que acompanharam junto comigo a jornada da graduação, Nathan, Jefferson, na qual, sempre me incentivaram a não desistir, continuar e sempre tentar. Aos meus amigos da vida, Bruno, Hugo, que nos momentos difíceis estavam lá para me fazer rir.

A Universidade Federal da Paraíba, que ofereceu oportunidades de participar de projetos e de estágio na conductor, dando início a minha carreira como analista de banco de dados.

As minhas amigas de trabalho, Cinthya e Thayane, na qual sempre conversamos sobre profissão, carreira e vida, o que tem feito eu não desistir dos meus objetivos como concluir a graduação.

“Qualquer retrato pintado com sentimento, é um
retrato não do modelo, mas do artista.”

O retrato de Dorian Gray
Oscar Wilde

RESUMO

Atualmente, na maioria das organizações empresariais e governamentais, é necessário utilizar alguma ferramenta que armazene e organize os dados, e com a crescente volumétrica dessas informações, no âmbito de desenvolvimento e aplicação, desenvolvedores e DBAs podem deparar-se com uma má performance em consultas SQL, que ocasiona a lentidão do sistema para a execução de tarefas. Nesse contexto, esta pesquisa discorre sobre o estudo de técnicas de performances aplicado a consultas nos bancos de dados relacionais, começando com o referencial teórico, abordando os conceitos gerais de banco de dados e os conceitos de algumas técnicas estudadas dentro do SQL Server, como o processo de tuning, a indexação e plano de execução. Os estudos de caso foram realizados considerando possíveis cenários de ocorrência tanto na produção quanto no desenvolvimento de dados. Foram feitos testes e comparações entre índices e operadores do SQL, com uma análise dos resultados, baseados nos planos de execução do SQL Server, no tempo decorrido e nos custos dos recursos utilizados.

Palavras-chave: Banco de Dados Relacional. Otimização. Performance. Tuning. SQL Server.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

BDs	Base de dados
DBAs	Administradores de Banco de Dados
MER	Modelo Entidade Relacionamento
NoSQL	Not Only SQL
PK	Primary Key (Chave primária)
SGBDs	Sistemas Gerenciadores de Bancos de Dados
SQL	Structured Query Language

SUMÁRIO

1. INTRODUÇÃO

Os Bancos de Dados (BD) são utilizados para a organização e armazenamentos de dados conforme as regras de negócio específicas, ou seja, cada base de dados tem seus próprios requisitos estabelecidos para algum objetivo final. Os banco de dados podem ser relacionais, que possui uma estrutura organizada dos dados que se relacionam entre si, ou não relacionais, que possui uma estrutura não estruturada dos dados.

Apesar do avanço dos BDs não relacionais, o uso de BDs relacionais é muito importante, muitas empresas e organizações utilizam bancos relacionais como ferramenta principal para armazenamento e desenvolvimento de dados.

O armazenamento de dados ocorre em diferentes tecnologias, em arquivos, em bancos de dados relacionais e em bancos de dados não relacionais (NoSQL). A grande disponibilidade de dados caracteriza uma situação contrária à do século XX, quando parte do problema era a ausência de dados em apoio à gestão (SORDI, 2019).

A disseminação do uso de computadores, tanto nas empresas quanto nos lares, contribuiu para o avanço da tecnologia de banco de dados, chegando ao ponto de assumir um papel importantíssimo em qualquer área que utiliza recursos informatizados (ALVES, 2014). Além disso, possui muitas vantagens, como a integridade e consistência dos dados. Essas características se dão através de chaves primárias e restrições. Praticidade, eficiência, rapidez na consulta e confiabilidade das informações foram os fatores principais que levaram ao desenvolvimento dos bancos de dados computadorizados (ALVES, 2014).

Importante destacar também a segurança de dados, os Sistemas Gerenciadores de Banco de Dados (SGBDs), oferecem aos administradores de bancos, recursos de permissão de usuário para controlar o acesso aos dados, restringindo a um grupo de usuários.

Nesse contexto, BDs relacionais que armazenam dados bancários, empresariais, comerciais, institucionais, governamentais, podem possuir milhões de dados de forma crescente, contendo uma grande escala de informações, o que pode dificultar o desempenho no desenvolvimento de dados. As aplicações que recuperam esses dados precisam ser eficientes para retorná-los de forma eficaz. Desta forma, a otimização das consultas é importante para garantir a rapidez no retorno das informações.

Diante do exposto, esse trabalho tem como objetivo principal, estudar as principais técnicas de otimização para BDs relacionais e aplicá-las em um estudo de caso, fazendo comparações entre tipos diferentes de cenários.

2. REFERENCIAL TEÓRICO

2.1 Conceitos gerais de Bancos de Dados Relacionais

Há algumas décadas atrás, o armazenamento de dados em empresas e órgãos públicos eram organizados e armazenados em papeis, divididos por grandes volumes de armários e pastas, o que dificultava o acesso a informações específicas. Com o avanço da computação e do grande volume de dados, se tornou imprescindível a existência de uma ferramenta de armazenamento mais prática, prezando a organização desses dados.

Um banco de dados relacional se caracteriza pelo fato de organizar os dados em tabelas (ou relações), formadas por linhas e colunas. Assim, essas tabelas são similares a conjuntos de elementos ou objetos, uma vez que relacionam as informações referentes a um mesmo assunto de modo organizado (ALVES, 2014).

O modelo de banco de dados relacional foi introduzido em 1970 por Edgar F. Codd em um artigo publicado pela IBM Research, intitulado “A Relational Model of Data for Large Shared Data Banks” (Modelo de dados relacional para grandes bancos de dados compartilhados), que gerou um marco na história dos BDs e se tornou uma abordagem muito utilizada para gerenciar dados, até os dias atuais.

Pelas palavras de C.J. Date (2014), essas ideias de Edgar F. Codd tiveram uma ampla influência em quase todos os aspectos da tecnologia de bancos de dados e também em outras áreas, como a inteligência artificial, o processamento da linguagem natural e o projeto de sistemas de *hardware*.

O modelo relacional é uma teoria abstrata de dados que se baseia em certos aspectos da matemática, principalmente na teoria dos conjuntos e na lógica de predicados (DATE, 2004). Em outras palavras, um modelo relacional é um conjunto de regras específicas para uma resolução de problemas ou para um negócio customizado, por exemplo, qualquer sistema que armazene informações tem que ter um ideia de como implementar baseado em um negócio definido.

Segundo uma das definições de Elmasri e Navathe, “Um banco de dados é projetado, construído e povoado por dados, atendendo a uma proposta específica. Possui um grupo de usuários definido e algumas aplicações pré concebidas, de acordo com o interesse desse grupo de usuários.” (ELMASRI & NAVATHE, 2005). Sendo assim, bancos de dados nada mais é que um conjunto de dados estruturados e organizados para um fim específico.

2.2 Modelagem de Bancos de Dados Relacionais

No que se refere à disponibilidade e valorização crescente dos dados, torna-se fundamental ter boa compreensão dos conceitos, das técnicas e dos métodos voltados à modelagem (especificação) dos dados. No ambiente tecnológico sempre há uma preocupação funcional, isto é, uma preocupação com o saber fazer (SORDI, 2019).

Nos ambientes corporativos e na criação de sistemas, a falta de uma clara definição de requisitos e regras de negócio resulta em informações desorganizadas. Consequentemente, quando chega a hora do desenvolvimento, a ausência de uma modelagem eficiente acaba gerando sistemas vulneráveis, inconsistentes e pouco eficazes. Essa falta de estrutura propicia uma série de manutenções corretivas para lidar com *bugs*, que poderiam ter sido evitadas se houvesse uma prévia definição de requisitos e uma modelagem adequada dos dados.

Com isso em mente, é fundamental que qualquer sistema em desenvolvimento adote a prática de modelagem de dados. Essa abordagem tem como principal objetivo garantir a manutenção, integridade e consistência dos dados armazenados nos sistemas. Além disso, no contexto da otimização, a fase de modelagem de dados assume extrema importância para prevenir a lentidão à medida que as bases de dados crescem. Com uma modelagem bem definida, é possível antecipar os possíveis cenários que levariam a um desempenho inadequado, frequentemente resultante de uma modelagem deficiente.

Dentre estes fatores, merece destaque a qualidade da estrutura do banco de dados. É necessário dar atenção ao seu desempenho desde a fase da modelagem, pois, a ausência ou mal uso de índices e a desnormalização das tabelas afetam a velocidade das consultas e atualizações, além de causar redundâncias e erros (MACHADO, 2014).

2.2.1 Modelo Entidade Relacionamento

Uma das dificuldades recorrentes enfrentadas pelos projetistas de banco de dados é representar toda a semântica que se encontra associada aos dados presentes no minimundo. O Modelo Entidade-Relacionamento (MER) foi criado justamente para procurar sanar essa deficiência. Ele é um modelo de dados de alto nível utilizado na fase de projeto conceitual, ou seja, na concepção do esquema conceitual do banco de dados (ALVES, 2019).

Para construir um modelo de dados, usa-se uma linguagem de modelagem de dados que podem ser classificadas de acordo com a forma de apresentar modelos, em linguagens textuais ou linguagens gráficas (HEUSER, 2011). Dessa forma, as linguagens citadas, têm a intenção de transformar uma ideia conceitual em algo que possa ser traduzido em termos computacionais.

Com a modelagem de dados é possível refinar um modelo conceitual durante as fases que compõem o projeto, eliminando redundâncias ou incoerências que possam inevitavelmente surgir.

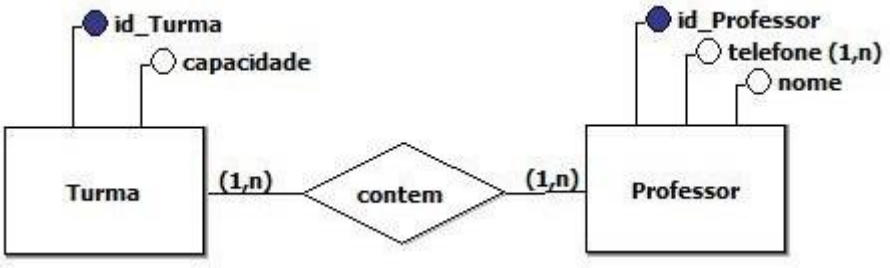
Sendo assim, o modelo MER é uma técnica de representar os atributos e relacionamentos entre entidades. Com essa definição, é importante começar com alguns conceitos básicos do MER, na qual, tem três elementos básicos: entidade, atributos e relacionamentos:

- Entidade: alguma coisa do mundo real ou virtual, um ser, uma pessoa, um evento, uma empresa que é representado por suas características. Exemplo: entidade “Aluno”;
- Atributos: características ou propriedades das entidades. Exemplo: nome, data de nascimento de uma entidade “Aluno”;
- Relacionamentos: representa as relações entre entidades de como são conectadas entre si. Exemplo: a conexão entre as entidades “Aluno” e “Disciplina”;
- Cardinalidade: Define o número de ocorrências de uma entidade que podem estar relacionadas a outra entidade. Exemplo, em um relacionamento entre “Aluno” e “Disciplina”, a cardinalidade indica que um aluno pode ter várias disciplinas (um-para-muitos) ou apenas uma disciplina (um-para-um).

2.2.2 Modelo Conceitual

Para efetivamente descrever as informações de um banco de dados, é recomendado começar pelo modelo conceitual. Este estágio inicial retrata, de forma próxima à realidade, as entidades, seus atributos e os relacionamentos entre elas. Geralmente apresentado por meio de desenhos, gráficos ou diagramas, o modelo conceitual proporciona um espaço abstrato e informal para o *brainstorming* de ideias e requisitos. Abaixo, segue uma figura ilustrativa representando o modelo conceitual:

Figura 1 - representação de um modelo conceitual



Fonte: Modelagem de Bancos de Dados: Conceitual, Lógica e Física (spaceprogrammer.com)

2.2.3 Modelo Lógico

De acordo com Heuser (2009) e Machado (2014), o modelo lógico descreve e mapeia as estruturas que estarão presentes no banco de dados, de acordo com as características da abordagem.

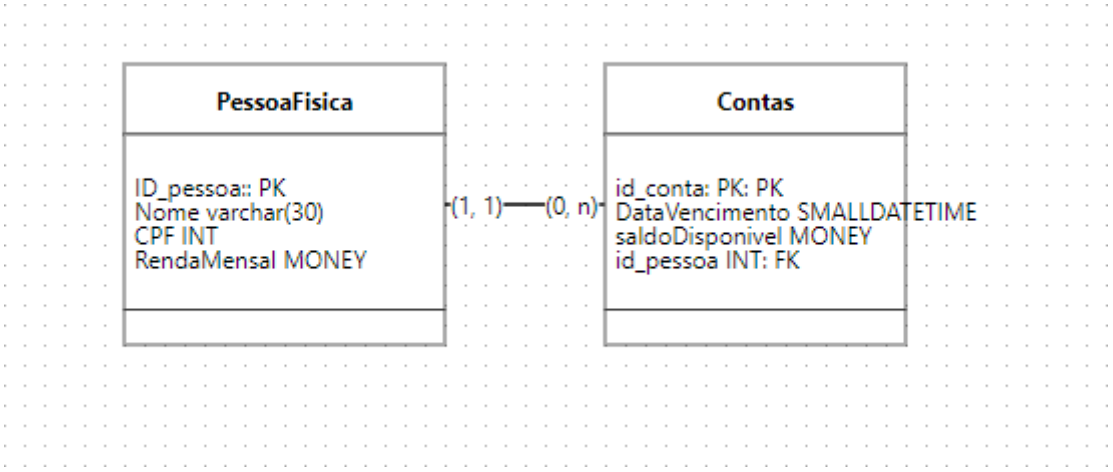
Evitam-se:

- Muitas tabelas;
- Tempo longo de resposta nas consultas e atualizações de dados;
- Desperdício de espaço;
- Muitos controles de integridade no banco de dados;
- Muitas dependências entre dados.

Com isso, o modelo lógico de dados é uma representação mais detalhada e específica do modelo conceitual, são descritas as tabelas que serão utilizadas junto com os tipos de dados dos campos (atributos). Inclui também, as chaves primárias e estrangeiras de cada tabela, as restrições necessárias dos atributos, as especificações de índices e a normalização.

A seguir, mostraremos um exemplo de modelo lógico para representar o banco de dados que será utilizado neste trabalho. A entidade (tabela) “Pessoa Física”, representada por seus atributos (campos) e tipos de dados, que se relacionam com a entidade (tabela) “Contas”, que contém o chave estrangeira “id_pessoa” e os outros atributos.

Figura 2 – Modelo criado no BRModelo



Fonte: autoria própria.

2.3 Sistema Gerenciadores de Bancos de Dados

Os sistemas gerenciadores de bancos de dados (SGBDs) são softwares que auxiliam no desenvolvimento e entendimento das estruturas dos bancos de dados, também gerenciam grandes volumes de dados, garantindo sua segurança e acessibilidade para consultas e modificações. Os SGBDS possuem uma interface CRUD (*create, read, update, delete*) onde o usuário é capaz de realizar leitura, atualização, modificação e criação. São também os SGBDs que proporcionam os caminhos de realizar consultas ao BD usando linguagem como o SQL.

Segundo Silberschatz (2020), uma das principais razões para usar os SGBDs é ter um controle central sobre os dados e sobre os programas que os acessam. Outrossim, Silberschatz e Sudarshan (2016), citam que o principal objetivo de um SGBD é proporcionar uma forma de armazenar e recuperar informações de um banco de dados de maneira conveniente e eficiente.

A principal vantagem do uso de um sistema de gerenciamento de banco de dados é o estado coerente dos dados que se encontram armazenados no banco. Isso faz com que as informações extraídas dele sejam confiáveis e de grande credibilidade (ALVES, 2014).

Os SGBDs vão ser de grande importância para a discussão do tema de otimização, pois possuem muitas ferramentas disponíveis dependendo do fabricante/desenvolvedor. Segundo uma pesquisa do “Stack Overflow Developer” os SGBDs mais populares entre profissionais são:

- SQL Server;
- PostgreSQL;
- Mysql;
- MongoDB;
- Oracle.

2.4 Tabelas

As tabelas são estruturas fundamentais dos bancos de dados, pois organizam dados de forma prática que facilitam a consulta de informações. Consistem em linhas e colunas, como uma espécie de planilha, na qual cada campo é uma coluna definida por um tipo de dado. Para facilitar o entendimento, criamos uma base de dados de uma empresa fictícia de uma loja de roupas, contendo uma tabela chamada “Clientes” com as seguintes informações:

Tabela 1 - Exemplo de tabela com colunas e linhas

	ID	Nome	Nascimento	RendaMensal	StatusAtivo	CPF
1	1	João Silva	1980-05-15	3500,00	1	123456789
2	2	Maria Santos	1992-08-20	2800,50	1	987654321
3	3	Pedro Almeida	1985-03-10	4500,75	1	567890123
4	4	Ana Oliveira	1998-12-05	2200,25	0	876543210
5	5	Carlos Mendes	1976-11-25	5000,00	1	345678901

Fonte: autoria própria

A imagem acima mostra um exemplo de uma estrutura de tabela, com colunas e linhas. Nas colunas ficam os campos com um nome de identificação único que representam diferentes tipos de dados já conhecidos. Dessa forma, o campo “Nome” é associado ao tipo de dado varchar, sendo uma cadeia de caracteres, o campo “ID” do tipo de dado INT, que contém apenas números inteiros, e assim por diante.

As tabelas vão ser cruciais no estudo de otimização, pois são as fontes principais de retorno dos dados, visto que algumas técnicas estudadas são aplicadas diretamente na estrutura dos campos das tabelas.

2.5 Chave Primária

As chaves primárias são colunas que contêm valores únicos para identificar cada registro na tabela, como mostrado no campo “ID”, na tabela 1 do subtópico anterior. As chaves primárias não podem se repetir, tornando possível consultar os dados de forma precisa, como se fosse o registro único de cada cidadão, CPF e RG, pois só existe um que é referente a um indivíduo.

Quando se cria uma chave primária, por padrão, está sendo criado também um índice *clustered*, isto é, um índice que organiza a sequência de gravação dos registros dentro dos arquivos físicos. Esta organização é de fundamental importância para a boa performance, pois facilita a localização física dos registros (CRIVELINI, 2013).

Sendo assim, caso a tabela não tenha nenhuma chave primária, os registros serão gravados em disco sequencialmente, na mesma ordem em que são inseridos, sendo uma das causas do mau desempenho.

2.6 Linguagem SQL

SQL (*Structured Query Language*) é uma linguagem de programação usada para gerenciar, manipular e consultar bancos de dados relacionais. De acordo com a definição de Silberschatz (2006) “Embora a SQL seja referenciada como ‘linguagem de consulta’, ela pode fazer muito mais do que simplesmente consultar um banco de dados, pois ela consegue definir a estrutura dos dados, modificar dados no banco de dados e especificar restrições de segurança.” Além disso, ela foi projetada para ser uma linguagem padrão que permite aos desenvolvedores interagir com sistemas de gerenciamento de banco de dados (SGBDs) para criar, modificar e consultar dados.

Os comandos SQL podem ser organizados em cinco categorias:

1. DDL - *Data Definition Language*: comandos usados para definir o esquema do banco de dados, ou seja, comandos que criam, modificam e deletam estruturas das bases de dados e não os dados em si.

- CREATE;
- DROP;
- ALTER;
- TRUNCATE.

1. DQL – *Data Query Language*: Representa a principal instrução do SQL, possui apenas um comando, o SELECT, que realiza as consultas nas bases de dados.
2. DML – *Data Manipulation Language*: Comandos que lidam com a manipulação dos dados presentes nos bancos de dados, inserindo, atualizando e deletando dados.

- INSERT;
- UPDATE;
- DELETE.

3. DCL – *Data Control Language*: Comandos que lidam com a permissão e controle de usuários nos bancos.

- GRANT;
- REVOKE.

4. TCL – *Transaction Control Language*: Comandos que possuem controle das transações, podendo reverter os comandos.

- COMMIT;
- ROLLBACK;
- BEGIN TRAN.

2.7 Procedimentos Armazenados e Gatilho

Procedimentos Armazenados ou *Stored Procedures*, é um conjunto de comandos em *script* SQL usado para fazer execuções DML (*insert, delete e update*). São utilizados quando há repetições de tarefas, como por exemplo, uma *procedure* que atualiza o saldo de uma conta bancária.

Vejamos alguns tipos de Procedimentos Armazenados:

- Procedimentos locais: criado pelo próprio usuário no banco de dados local;
- Procedimentos temporários: procedimentos locais e procedimentos globais;
- Procedimentos do sistema: procedimentos que vêm armazenados por padrão nas SGBDs, são como funções específicas do sistema que podem ser utilizadas em qualquer banco de dados. Exemplo: função "*sp_helptext*" que retorna o escopo de um objeto.

Já um Gatilho ou *Trigger*, é uma *procedure* que vai desencadear uma ação quando for acionada dependendo da configuração feita pelo usuário. Como por exemplo, monitorar uma tabela importante de transação e toda vez que houver eventos de *Insert, Delete ou Update*, essas informações serão gravadas imediatamente em uma outra tabela de *log*, com objetivo de controlar e recuperar esses dados.

Sendo assim, um gatilho é acionado toda vez que houver um evento, podendo ser eventos de *Insert, Delete ou Update*, quanto alguma configuração programada. Esse mecanismo é uma ferramenta de ação automática muito importante para o desenvolvedor e DBA, capaz de otimizar tarefas, obter controle de dados, automatizar processos, realizar operações de auditoria e impedir transações inválidas.

2.8 Técnicas de Otimização

Atualmente, em grandes empresas do mercado que utilizam serviços de pagamentos, transações bancárias, processamento de transações e compras *online*, há uma dependência exacerbada do uso de sistema de banco de dados, exigindo uma disponibilidade integral. A indisponibilidade desses serviços se dá pelo mau desempenho dos bancos de dados, por exemplo, quando ocorre algum problema relacionado a lentidão, pode impactar diretamente no usuário final, podendo gerar grandes perdas financeiras às empresas.

2.8.1 Principais Causas de Problemas de Desempenho em Banco de Dados

O mau desempenho de um sistema de banco de dados pode ser relacionado ao servidor e a sua arquitetura física, mas segundo Picini e Schimiguel (2008), as principais causas são:

- 60% dos problemas estão relacionados ao mau uso de expressões SQL;
- 20% dos problemas estão relacionados à má modelagem do Banco;
- 10% dos problemas estão relacionados à má configuração do SGDB;
- 10% dos problemas estão relacionados à má configuração do S.O.

Já Mullins (1999), afirma que, quase 80% dos problemas de performance em banco de dados podem ser encontrados em códigos SQL mal elaborados.

Além dessas hipóteses, os autores Fritchey e Dam (2014) destacam outros motivos, como:

- Falta de indexação: sem uma boa estratégia de indexação, uma consulta leva mais tempo e recursos para ser utilizada, já que o SGBD vai precisar processar mais dados para retornar o resultado.
- Estatísticas Imprecisas**:** para que o uso dos índices seja efetivo, é necessário manter estatísticas precisas e atualizadas sobre a quantidade de linhas recuperadas nas consultas. Estes dados são importantes para que o otimizador de consultas do SQL Server decida a melhor estratégia de otimização. Sem dados precisos, sua eficiência é prejudicada (OLIVEIRA; PAIVA, 2018).

Tendo em vista os pontos citados, a otimização de consultas em bancos de dados é um trabalho extremamente necessário e importante para desenvolvedores e DBAs, na qual, tem a finalidade de mostrar ao usuário a informação solicitada em menor tempo possível. No contexto prático do mercado, esperar muito tempo por uma consulta, seja ela em *procedures* ou *triggers*, não é viável.

Existem diversos tipos de otimização que são utilizados, mas é importante entender o problema existente e adequar a melhor técnica de solução na situação específica. Adiante, iremos expor algumas técnicas de otimização.

2.8.2 Sintonização (*tuning*)

O *tuning* de SQL tem o objetivo de auxiliar na escrita e na execução adequada de comandos SQL, que permite um maior desempenho da manipulação de dados. O monitoramento e o ajuste destes comandos irão proporcionar ganhos para o sistema, evitando problemas de falta de performance (GONÇALVES, 2006). Embora o *tuning* de SQL seja um conceito aplicável a diversos SGBDs, como MySQL, Oracle, PostgreSQL, entre outros, cada SGBD possui suas próprias ferramentas e técnicas específicas para otimização.

O sintonizador pode, no entanto, modificar a tabela, projetar e selecionar novos índices, reorganizar transações e adulterar o funcionamento do sistema. Os objetivos são eliminar gargalos, diminuir o número de acessos aos discos e garantir tempo de resposta, pelo menos no sentido estatístico (SHASHA, 1996). Este processo é aplicável a diversos SGBDs, incluindo o SQL Server, o que demonstra a generalidade das técnicas de *tuning* de SQL.

Entendemos *tuning* da base de dados como uma customização do sistema sob medida, para que a performance atenda melhor às suas necessidades. O planejamento para o gerenciamento da performance do banco de dados é um componente crucial de qualquer implementação de aplicação. Sem um plano para monitorar performance e ajustar o banco de dados, a degradação da performance fatalmente ocorrerá. Um plano completo de gerenciamento de performance incluirá ferramentas para ajudar a monitorar a performance da aplicação e o ajuste do SGBD (IKEMATU, 2009).

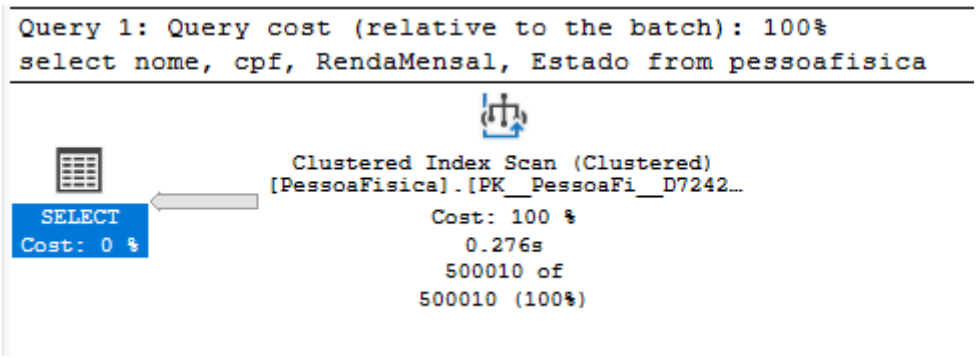
Em resumo, a sintonização ou *tuning* é uma técnica utilizada para otimizar o sistema de banco de dados, capaz de melhorar o desempenho, diminuir tempo de respostas das consultas, aumentar a velocidade das aplicações, a concorrência entre as transações, reduzir a sobrecarga de registro de *logs* e em identificar por meio de ferramentas de análise e estatística, como SQL Profile, possíveis gargalos no sistema.

2.8.3 Planos de Execução

Quando uma consulta é executada no SQL Server, como um SELECT, por exemplo, vários processos são executados ao mesmo tempo, e os planos de execução mostram as informações do uso dos recursos do sistema, seguindo um conjunto de instruções para executar uma consulta específica, detalhando como as tabelas estão sendo acessadas, como e quais índices serão utilizados, quantas páginas de dados serão lidas e qual operação teve mais custo de tempo. Ou seja, mostra as informações detalhadas dos recursos que a consulta utilizou do SQL Server, o que torna muito útil para ajudar a descobrir a melhor estratégia de melhorias que podem ser feitas, se baseando no resultado do plano. Por isso, é importante entender e analisar os planos de execução.

A figura 3 mostra um exemplo de detalhes do plano de execução, uma consulta simples a uma tabela que contém informações de endereços de pessoas. Contém informação do tipo de índice, estimativas de custo de operações de entrada e saída, custo da subárvore, estimativa de número de linhas afetadas que serão lidas durante a execução (conceitos que serão vistos nos próximos tópicos).

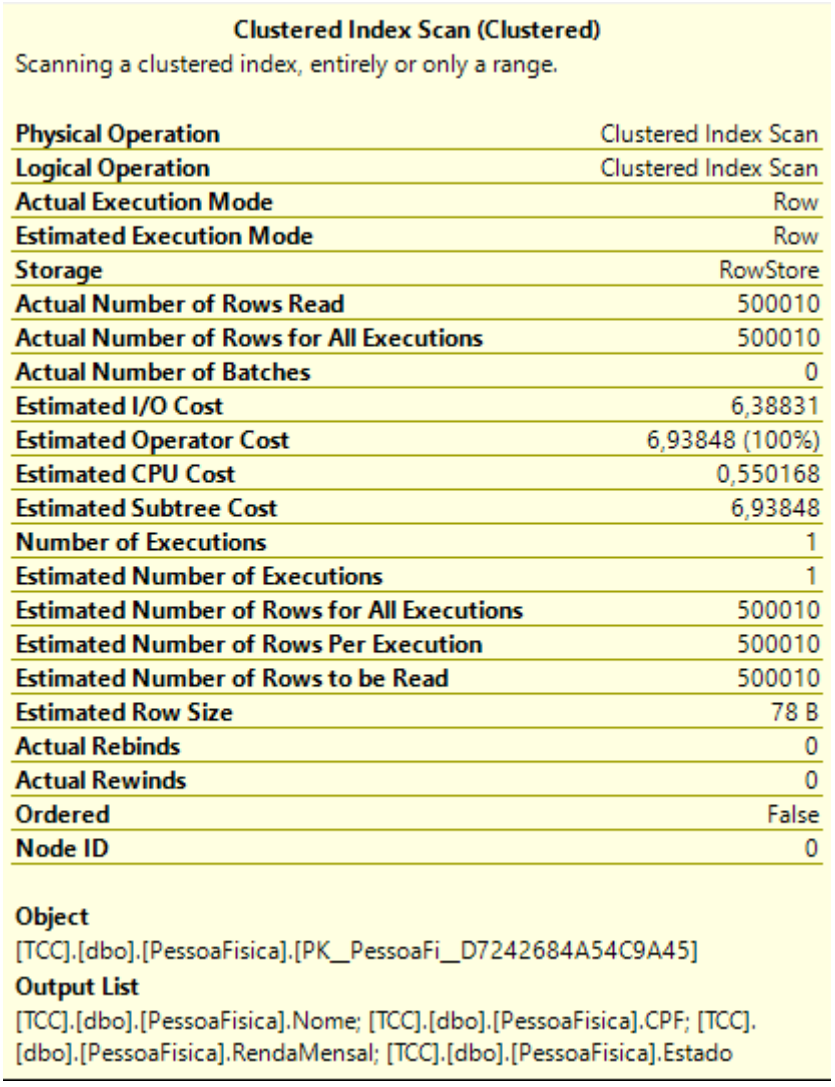
Figura 3 - Exemplo de um plano de execução



Fonte: autoria própria.

Ao passar o cursor em cima do Clustered Index Scan, o plano traz as informações detalhadas dos recursos como o tipo de operação, o custo do CPU, custo de I/O e número de linhas lidas:

Figura 4 - Detalhes do plano de execução



Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows Read	500010
Actual Number of Rows for All Executions	500010
Actual Number of Batches	0
Estimated I/O Cost	6,38831
Estimated Operator Cost	6,93848 (100%)
Estimated CPU Cost	0,550168
Estimated Subtree Cost	6,93848
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	500010
Estimated Number of Rows Per Execution	500010
Estimated Number of Rows to be Read	500010
Estimated Row Size	78 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Object

[TCC].[dbo].[PessoaFisica].[PK__PessoaFi__D7242684A54C9A45]

Output List

[TCC].[dbo].[PessoaFisica].Nome; [TCC].[dbo].[PessoaFisica].CPF; [TCC].[dbo].[PessoaFisica].RendaMensal; [TCC].[dbo].[PessoaFisica].Estado

Fonte: autoria própria.

2.8.4 Plano de Execução Estimado e Plano de Execução Real

O plano de execução estimado não realiza a consulta, ele apenas provê uma estimativa de quais recursos, de quantas páginas de dados serão lidas, de quanto tempo custará aquela consulta, sendo apenas uma simulação do que esperar da consulta.

O plano de execução estimado é útil para uma variedade de propósitos, como visualizar o plano de uma consulta de longa duração sem esperar que seja concluída, visualizar plano para

uma instrução de inserção, atualização ou exclusão sem ter que alterar o estado do banco de dados. (DELANEY, 2005).

O plano de execução real é executado junto com a consulta, ele mostra o que aconteceu durante o processamento da execução, trazendo os dados reais do que cada recurso utilizado custou, ou seja, traz as mesmas informações do estimado, mas adicionando os números reais de linhas afetadas e o número real de execução para cada operador.

2.8.5 Índices

Os índices podem ser utilizados como uma lista de contatos, nos quais os dados são armazenados fisicamente na ordem em que você adiciona as informações de contato das pessoas, porém é mais fácil encontrar pessoas quando listadas em ordem alfabética. (BARNHILL, 2019).

Podemos citar a lista telefônica como um exemplo de utilização de índices. Supondo que é uma lista enorme com muitos nomes e números, sem um índice fica difícil achar qualquer informação, tendo que percorrer página por página e linha por linha para encontrar o número de telefone desejado. Neste caso, com o índice organizado alfabeticamente ou por categorias, o problema de consulta de dados seria resolvido de forma prática.

Em um banco de dados, os índices armazenam informações sobre os dados na tabela organizados em páginas de dados, permitindo uma localização rápida do registro solicitado. Seguidamente, será analisada, de forma detalhada, a varredura feita pelo SQL Server e os tipos de índices.

2.8.6 Tipos de índices e leitura de dados

Existem alguns tipos de índices e leitura de dados que podem ser encontrados durante uma consulta. Há dois tipos de iteradores que o SQL Server usa para fazer leitura de dados de tabelas, eles aparecem em quase todos planos de execução e é importante entender a suas diferenças, vejamos:

- Index Scan: pode processar uma tabela inteira ou um nível de uma folha de um índice, busca os elementos da tabela de forma sequencial, um por um, até encontrar o elemento. O

custo dessa operação é proporcional ao número de linhas na tabela, ou seja, se for uma tabela de muitos registros o desempenho será afetado significativamente.

- Index Seek: ocorre quando o banco de dados utiliza o índice para localizar diretamente os registros que correspondem aos critérios da consulta, sem precisar percorrer todas as linhas da tabela através do índice definido. Procurando diretamente pela informação, é o melhor tipo de cenário que pode ocorrer fazendo uma busca binária. O custo dessa operação também é proporcional ao número de linhas na tabela, como vai ler menos linhas lidas, é mais eficiente.

Com isso, o SQL Server traz dois tipos de índices possíveis baseados nesses iteradores, sendo eles:

- Índice clusterizados: armazena todas as colunas da tabela em um índice no nível folha, ou seja, é a própria tabela, sendo a própria chave primária, garantindo que seja armazenada em ordem crescente, que também é a ordem que a tabela mantém na memória. Um índice clusterizado determina a ordem em que os registros são armazenados na tabela, não se tratando de um índice separado da tabela (JORGENSEN et al., 2012).
- Índices não-clusterizados: são índices criados para um campo específico da tabela principal, que retém ponteiros para as entradas originais da tabela. Para cada consulta nos registros será gerada uma subárvore para o campo, por exemplo, se desejar consultar uma conta pelo nome, o índice não-clusterizado irá fazer uma busca binária indo até a página folha, e como é ordenado, começa pela letra inicial, trazendo a informação com menos custo e não sendo necessário percorrer várias folhas desnecessárias. 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 (RAMALHO, 2005).

Na figura 5, é representada como Scan e Seek aparecem no plano de execução.

Figura 5 - Tipos de operadores

Operadores	Scan	Seek
Heap	Table Scan	
Clustered Index	Clustered Index Scan	Clustered Index Seek
NonClustered Index	Index Scan	Index Seek

Fonte: autoria própria.

Uma tabela *heap* no SQL Server representa uma tabela sem índices clusterizados, os dados são armazenados sem especificar nenhuma ordem, toda busca será sequencial e sempre será feita a leitura dos dados sendo definido como Table Scan.

De acordo com a documentação da Microsoft:

> Às vezes, há boas razões para deixar uma tabela como heap em vez de criar um índice clusterizado, mas usar heaps efetivamente é uma habilidade avançada. A maioria das tabelas deve ter um índice clusterizado cuidadosamente escolhido, a menos que haja uma boa razão boa para deixar a tabela como heap. (MICROSOFT, 2022)

O Clustered Index Scan é referente a chave primária da tabela e ocorre quando a consulta, de alguma forma, não utiliza o campo indexado, se a consulta a ser analisada utilizar outros campos não indexados, o clustered index scan vai ocorrer, então, o SQL Server vai retornar os registros da consulta de forma sequencial, é melhor que o table scan, mas ainda é uma busca sequencial.

O Clustered Index Seek, ocorre quando utiliza o campo desse índice na consulta, o SQL Server usa o índice clusterizado para localizar linhas específicas em uma tabela. Isso acontece quando a consulta inclui uma cláusula WHERE ou JOIN que corresponde exatamente à ordem de indexação do índice clusterizado.

O NonClustered Index *Scan* acontece quando a consulta não corresponde exatamente à ordem de indexação do índice não clusterizado,ocorre no SQL Server para escanear um índice não clusterizado em busca das linhas desejadas, quando o acesso direto às linhas da tabela não é eficiente e o uso do índice pode melhorar o desempenho da consulta.

O NonClustered Index Seek é o melhor cenário possível, usada pelo SQL Server para buscar, diretamente, as linhas desejadas em uma tabela usando um índice não clusterizado, quando a consulta corresponde exatamente à ordem de indexação do índice e o uso do índice pode melhorar o desempenho da consulta. Esses eventos serão mostrados no caso de uso.

2.8.7 Árvore *B-Trees*

A *B-Tree* é uma árvore balanceada, ou seja, todas as suas folhas[1] estão no mesmo nível. Ela é usada para armazenar grandes quantidades de dados em disco e em memória secundária, o que a torna muito útil em bancos de dados e sistemas de arquivos que manipulam conjuntos grandes de dados. Ela tenta equilibrar o custo de processamento com o custo de I/O.

As páginas de dados são tipos de armazenamento no nível físico de um banco de dados. Cada página tem um tamanho fixo. No SQL Server, os dados são organizados nessas páginas de dados e cada página possui 8 *kbytes*, ou seja, possui muitas páginas contendo os registros das tabelas, e o acesso a essas páginas se dá por meio dos índices.

Sendo assim, o SQL Server guarda as informações dos índices numa estrutura chamada *B-tree*, é muito utilizada em bancos de dados e sistemas de arquivos para organizar e conseguir acessar rapidamente as informações solicitadas.

Os índices afetam as operações de *update, insert e delete* e nem sempre o seu uso terá um bom desempenho. Se a tabela em questão tiver muitas operações de CRUD, será necessário reajustar as páginas de dados frequentemente. Dessa forma, não é favorável ter vários índices nela. Mas se a tabela for voltada para consulta e recuperação de dados, pode ser uma boa opção.

2.9 SQL Server

O Microsoft SQL Server é um gerenciador de bancos de dados relacional desenvolvido pela Sybase em parceria com a Microsoft, lançado em 24 de Abril de 1989. Sua linguagem é o T-SQL (SQL com algumas extensões e recursos avançados que facilitam o desenvolvimento).

Como citado no tópico 2.6, o SQL Server é um dos SGBDs mais populares entre os desenvolvedores. Ele possui versões pagas e gratuitas, sendo a edição gratuita chamada SQL Server Express, ideal para desenvolvimento em pequenos servidores e pequenas aplicações *Web*.

Por ser um gerenciador de banco de dados relacional, seus dados e informações estão armazenados em tabelas com linhas e colunas, podendo ser manipulados através de procedimentos, gatilhos e *views*. Dessa maneira, escolhemos o SGBD SQL Server da Microsoft como foco principal da nossa pesquisa.

[1] Em uma B-Tree, as folhas são os nós finais da árvore, que não possuem filhos. Elas contêm os apontadores para os registros reais do banco de dados ou para os blocos de dados nas estruturas de armazenamento.

3. TRABALHOS RELACIONADOS

Os bancos de dados precisam de um estudo dedicado à otimização de consultas e técnicas. Visto isso, Gava (2010) afirma que a eficiência e o bom desempenho de consultas são cruciais para o funcionamento dos sistemas, pois impacta diretamente em grandes empresas bancárias. Considerando isso, o referencial teórico deste estudo destaca as técnicas de otimização mais comuns, como índices, planos de execução e sintonização (*tuning)*. Tendo em vista esses fatores, Gava (2010) faz um estudo sobre essas três técnicas em seu trabalho, explicando os conceitos, justificando a importância de realizar o processo de otimização em bancos de grandes volumes de dados e como escolher a melhor estratégia, dependendo do problema.

O estudo de Rodrigues (2019) traz como objetivo a redução do tempo de resposta nos sistemas de BDs, na qual, acredita que o maior desafio hoje é manter a eficácia das aplicações em sistemas de gerenciamento de banco de dados, devido à grande quantidade de dados. Com isso, Rodrigues (2019) foca principalmente na utilização de índices em tabelas e planos de execução para analisar os resultados, sendo assim, através de testes e resultados com estatísticas, ele mostra como o uso de índices em tabelas melhora significativamente o desempenho de consultas.

No estudo de caso de Cabral, Corado e Bruno (2015), é exposto outros tipos de técnicas de otimização, nas quais são mais pontuais e consideradas de menor impacto, ou seja, geram mudanças mínimas no código. O estudo ainda compara operadores e comandos, como na junção de tabelas, além de relacionar INNER JOIN e WHERE em uma subconsulta, nos comandos que produzem *sort* (*group by, order by, distinct*), comparando uma consulta usando o comando *order by* a outra, sem o uso do *order by*. Por fim, os autores mostram os resultados obtidos com o uso do índice.

No artigo de Gonçalves (2006) é posto uma discussão a respeito da utilização dos índices, destacando que se não for utilizado corretamente poderá causar perda de desempenho. As operações de *Insert, Delete* e *Update* são as mais afetadas pelo uso incorreto dos índices, pois cada vez que uma linha é inserida em tabelas com colunas indexadas, é inserida também, uma linha no índice, podendo causar lentidão. Gonçalves (2006) alerta que cada caso pode haver ganho no retorno de linhas mas possuir efeitos colaterais em comandos DML.

4. METODOLOGIA

4. 1 Base de dados

Existem vários bancos de dados que requerem um número excessivo de atividade, e quando nos referimos a transações de compras, deduzimos que há uma grande sobrecarga de fluxo de dados. Visto isso, surgiu a necessidade de analisar de forma mais aprofundada, a principal funcionalidade do banco de dados de um banco financeiro, que é processar transações de compras feitas com cartões de crédito.

Dessa forma, decidimos criar uma base de dados fictícia de um sistema de gerenciamento de transações de cartão de crédito. Para isso, foi pensado um cenário de banco financeiro para realizar os testes, considerando as regras específicas de um banco, no contexto em que transações são geradas a todo momento em grande escala.

O banco de dados terá um nome fictício chamado “ViBank” e possui mais de 500 mil clientes. O banco financeiro fornece alguns produtos como o cartão de crédito e uma conta corrente conjunta de pessoas físicas.

- Cartões de Crédito: oferece aos clientes um cartão que faça compra a crédito;
- Contas Correntes: oferece aos clientes uma conta para realizar transações diárias, como depósitos, saques, transferências e pagamentos.

Nosso foco aqui, é em como otimizar *queries* que são mais recorrentes em um ambiente de desenvolvimento. Importante destacar que, a população de dados foi feita de forma fictícia, inserindo dados automaticamente nas tabelas, através de um script desenvolvido em SQL com dados aleatórios, não havendo correlação com nenhum banco existente. No entanto, vale salientar que, o método proposto é aplicável a outros bancos de dados.

No sistema em questão, as principais tabelas são: “PessoaFisica”, que especifica os dados pessoais do cliente; “Conta”, que é referente a conta do cliente, como saldo disponível e status da conta; “Cartoes”, onde constam dados do saldo da fatura, vencimento e status. E a tabela “Transacoes”, onde ficará todas as transações de compras feitas por cada cliente, o valor, data da compra e nome do estabelecimento.

Apesar da modelagem ter sido especificada desde o começo, há tabelas que não possuem índices bem definidos, e tabelas que foram definidas serão alteradas para mostrar os resultados das diferenças entre os tipos de índices, apresentados no tópico anterior.

Foram inseridos 500 mil registros de pessoas físicas fictícias na tabela “PessoasFisicas”, representando todos os clientes ativos, e assim, para as outras tabelas que também fazem parte do sistema e conexão.

Alguns testes foram feitos pensando nos problemas mais comuns que poderiam ocorrer ao fazer uma consulta, como a lentidão em tabelas, que desencadeia vários outros problemas, principalmente em procedimentos armazenados e *triggers* que utilizam essas tabelas.

No nosso sistema, haverá uma *trigger* que é acionada toda vez que há alteração de *update, insert* e *delete* nos saldos do cartão e da conta, inserindo numa tabela chamada “CartoesLog”. O *log* é muito utilizado para rastrear possíveis erros de informações, além de não super popular a tabela principal. Essa tabela será utilizada para os testes, pois contém a maior parte de inserção de dados. Além disso, há um procedimento chamado “AtualizaConta”, que ficará responsável por atualizar os saldos, vencimentos e limites das contas nas tabelas principais.

A geração dos dados foi através do script abaixo, onde cria-se um *loop* com a quantidade de registros que se quer inserir, e a cada inserção são inseridos dados aleatórios de forma automática, vejamos no teste feito no ambiente do *SQL Server* em um notebook pessoal (processador I7, 8GB de memória RAM e 240GB de SSD):

```
DECLARE @i INT = 1;
WHILE @i <= 50000
BEGIN
   INSERT INTO PessoaFisica (id_pessoafisica, Nome, CPF, RendaMensal, Estado, Cidade)
   VALUES (ABS(CHECKSUM(NEWID())),
        CONCAT('Nome', ABS(CHECKSUM(NEWID())) % 1000),
        FORMAT(ABS(CHECKSUM(NEWID())) % 99999999999, '00000000000'),
        RAND() * 10000,
        CONCAT('Estado', ABS(CHECKSUM(NEWID())) % 100),
        CONCAT('Cidade', ABS(CHECKSUM(NEWID())) % 1000));
   SET @i = @i + 1;
END
```

A partir disso, criamos 3 casos de testes. O primeiro caso é sobre a indexação, comparando diferentes tipos de utilização, ou não, dos índices e como isso pode afetar a performance das consultas. O segundo teste é uma comparação entre junção de dados em uma consulta para trazer todos os dados. Por fim, o terceiro caso irá subtrair as tabelas, seguindo com os resultados dos testes.

5. CASOS DE TESTE

A tabela principal a ser testada e estudada será a “CartoesLog”, que possui o controle de *insert, delete* e *update* da tabela “Cartoes”, ou seja, toda alteração feita em “Cartoes” está registrado, como na tabela de log. A tabela em questão possui uma grande quantidade de dados, com cinco milhões e meio de registros, sendo possível trabalhar para mostrar os casos de otimização.

Além disso, por ser uma tabela de *log* e controle, é muito utilizada para consultas de SELECT, de retorno rápido de informações. Sendo assim, a análise comparativa será feita através dos planos de execuções mostrados no subtópico 2.8.3, que possui as estatísticas necessárias para observação.

As métricas utilizadas para gravar tempo de execução serão feitas através do comando SET STATISTICS TIME ON, no qual exibe o número de milissegundos necessários para analisar, compilar e executar cada instrução. Dessa forma, o comando irá retornar duas informações a respeito do tempo de análise e compilação do SQL Server e Tempos de Execução do SQL Server. O primeiro refere-se ao tempo gasto pelo SQL Server para analisar a consulta SQL e compilar um plano de execução para ela, verificando a sintaxe e validando os objetos. O segundo indica o gasto durante a execução real da consulta, incluindo tempo de CPU consumido pela consulta e o tempo decorrido desde o início até a conclusão da execução.

- Tempo de CPU: refere-se ao tempo total da CPU utilizado durante a execução da consulta, quanto maior tempo maior uso dos recursos foi utilizado.
- Tempo de execução: refere-se ao tempo total de execução contando com o tempo da cpu, tempo de outros recursos e possíveis bloqueios no geral.

São informações úteis para medir os detalhes e identificar possíveis empecilhos de uma consulta específica, por exemplo, em caso de uma análise de uma *procedure*, será identificado o ponto que está mais lento.

5.1 Teste 1: Utilização dos índices na prática

A fim de implementar e compreender a utilização dos índices descritos no referencial teórico, será feito comparações entre consultas mais comuns para acessar a tabela "CartoesLog" e "Cartoes", modificando-a para ser uma tabela sem índices (um índice *clustered* e um índice *nonclustered* em algum campo).

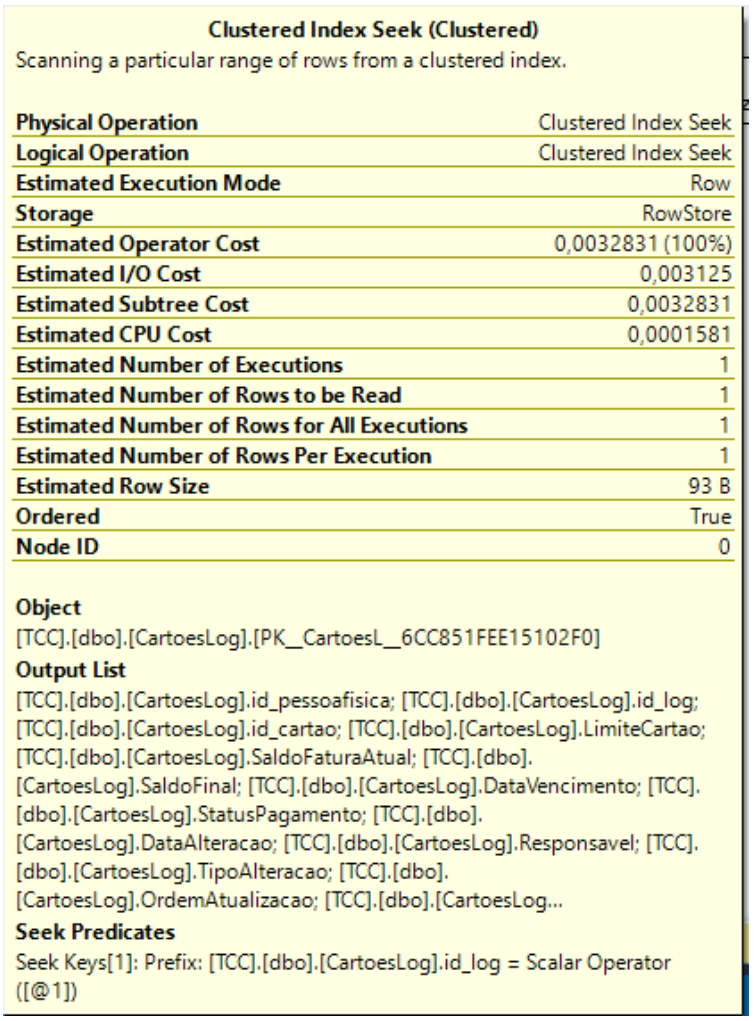
a) Índice *Clustered*

A tabela foi criada com uma chave primária no campo "Id_Log", tornando-a automaticamente como uma tabela com índice *clustered*. Aplicando a seguinte consulta de busca, obtém os seguintes resultados:

```
SELECT * FROM CartoesLog WHERE id_log = 2000038
```

O plano de execução traz a informação que foi feito uma varredura do tipo Clustered Index Seek, ou seja, a tabela utilizou seu índice "id_log" que é a PK e o índice *clustered*, fazendo uma busca binária. A figura abaixo mostra os detalhes dos custos da consulta.

Figura 6 - Clustered Index Seek



Clustered Index Seek (Clustered)

Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0,0032831 (100%)
Estimated I/O Cost	0,003125
Estimated Subtree Cost	0,0032831
Estimated CPU Cost	0,0001581
Estimated Number of Executions	1
Estimated Number of Rows to be Read	1
Estimated Number of Rows for All Executions	1
Estimated Number of Rows Per Execution	1
Estimated Row Size	93 B
Ordered	True
Node ID	0

Object
[TCC].[dbo].[CartoesLog].[PK__CartoesL__6CC851FEE15102F0]
Output List
[TCC].[dbo].[CartoesLog].id_pessoafisica; [TCC].[dbo].[CartoesLog].id_log; [TCC].[dbo].[CartoesLog].id_cartao; [TCC].[dbo].[CartoesLog].LimiteCartao; [TCC].[dbo].[CartoesLog].SaldoFaturaAtual; [TCC].[dbo].[CartoesLog].SaldoFinal; [TCC].[dbo].[CartoesLog].DataVencimento; [TCC].[dbo].[CartoesLog].StatusPagamento; [TCC].[dbo].[CartoesLog].DataAlteracao; [TCC].[dbo].[CartoesLog].Responsavel; [TCC].[dbo].[CartoesLog].TipoAlteracao; [TCC].[dbo].[CartoesLog].OrdemAtualizacao; [TCC].[dbo].[CartoesLog...
Seek Predicates
Seek Keys[1]: Prefix: [TCC].[dbo].[CartoesLog].id_log = Scalar Operator ([@1])

Fonte: autoria própria.

Resultados obtidos:

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 23 ms.

- Tempo de execução do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 0 ms.

  Linhas afetadas: 1

b) Campo sem índice

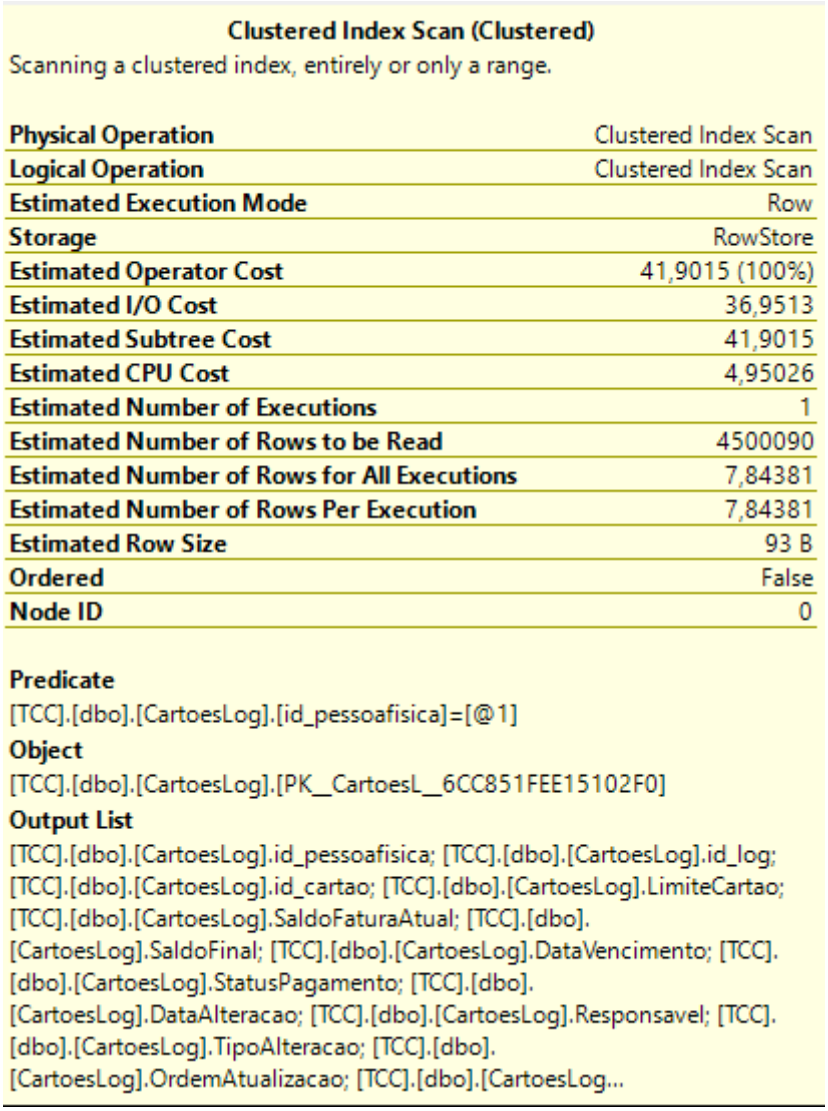
Em seguida, aplicaremos a mesma consulta, porém, utilizando outro campo na especificação do WHERE, o campo “id_pessoafisica”, no qual, não possui nenhum índice, vejamos:

```
SELECT * FROM CartoesLog WHERE id_pessoafisica = 639224389
```

Nesse caso, ocorreu o Clustered Index Scan, pois, não há nenhum índice especificado para esse campo na tabela, fazendo com que a varredura seja realizada, sequencialmente, em todas as páginas de dados.

.

Figura 7 - Clustered Index Scan



Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	41,9015 (100%)
Estimated I/O Cost	36,9513
Estimated Subtree Cost	41,9015
Estimated CPU Cost	4,95026
Estimated Number of Executions	1
Estimated Number of Rows to be Read	4500090
Estimated Number of Rows for All Executions	7,84381
Estimated Number of Rows Per Execution	7,84381
Estimated Row Size	93 B
Ordered	False
Node ID	0

Predicate
[TCC].[dbo].[CartoesLog].[id_pessoafisica]=[@1]
Object
[TCC].[dbo].[CartoesLog].[PK__CartoesL__6CC851FEE15102F0]
Output List
[TCC].[dbo].[CartoesLog].id_pessoafisica; [TCC].[dbo].[CartoesLog].id_log; [TCC].[dbo].[CartoesLog].id_cartao; [TCC].[dbo].[CartoesLog].LimiteCartao; [TCC].[dbo].[CartoesLog].SaldoFaturaAtual; [TCC].[dbo].[CartoesLog].SaldoFinal; [TCC].[dbo].[CartoesLog].DataVencimento; [TCC].[dbo].[CartoesLog].StatusPagamento; [TCC].[dbo].[CartoesLog].DataAlteracao; [TCC].[dbo].[CartoesLog].Responsavel; [TCC].[dbo].[CartoesLog].TipoAlteracao; [TCC].[dbo].[CartoesLog].OrdemAtualizacao; [TCC].[dbo].[CartoesLog...

Fonte: autoria própria.

Resultados obtidos:

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 22 ms.

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 734 ms, tempo decorrido = 1346 ms.
    Linhas afetadas: 9

c) Índice *NonClustered*

Com isso, torna-se necessário a criação de um índice *nonclustered* no campo "id_pessoafisica", pois será mais utilizado do que a "id_log" nesse tipo de caso. Vejamos o comando de criação:

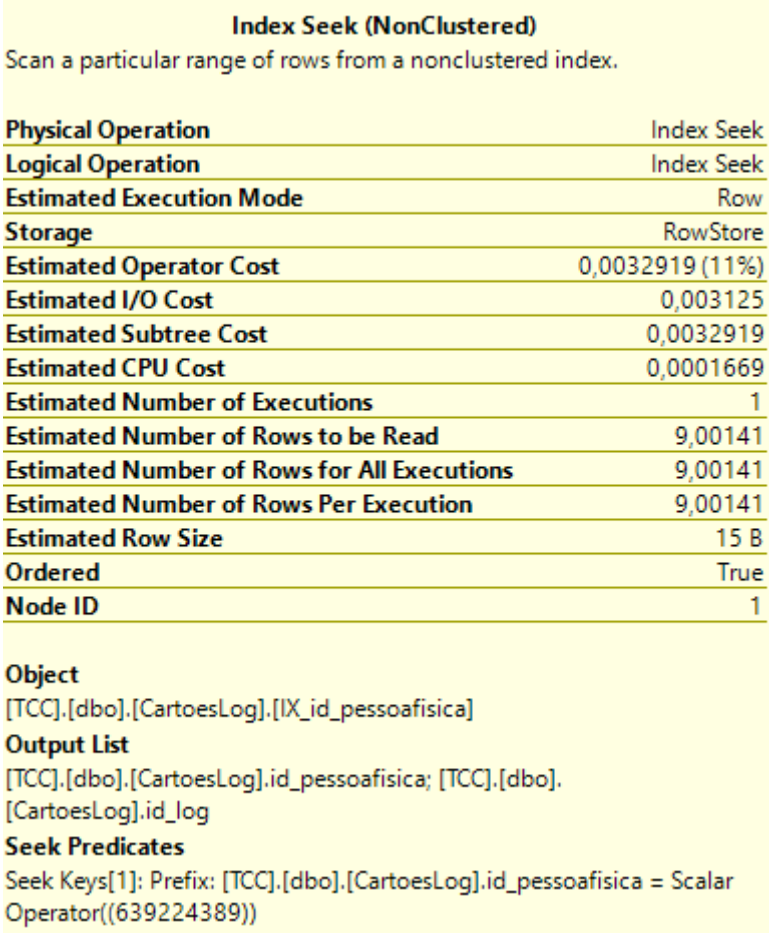
```
CREATE NONCLUSTERED INDEX IX_id_pessoafisica ON CartoesLog (id_pessoafisica)
```

Após a criação de índice e aplicando novamente a consulta, obtivemos:

```
SELECT * FROM CartoesLog WHERE id_pessoafisica = 639224389
```

Dado isso, deduzimos que houve melhora para Index Seek, ou seja, a busca para o campo "id_pessoafisica" especificado com índice foi mais rápido em relação aos outros casos que contém índices. Ao invés de percorrer linha por linha, a busca foi imediatamente para as páginas armazenadas como uma árvore.

Figura 8 - Index Seek



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0,0032919 (11%)
Estimated I/O Cost	0,003125
Estimated Subtree Cost	0,0032919
Estimated CPU Cost	0,0001669
Estimated Number of Executions	1
Estimated Number of Rows to be Read	9,00141
Estimated Number of Rows for All Executions	9,00141
Estimated Number of Rows Per Execution	9,00141
Estimated Row Size	15 B
Ordered	True
Node ID	1

Object
[TCC].[dbo].[CartoesLog].[IX_id_pessoafisica]
Output List
[TCC].[dbo].[CartoesLog].id_pessoafisica; [TCC].[dbo].[CartoesLog].id_log
Seek Predicates
Seek Keys[1]: Prefix: [TCC].[dbo].[CartoesLog].id_pessoafisica = Scalar Operator((639224389))

Fonte: autoria própria.

Resultados obtidos:

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 16 ms, tempo decorrido = 18 ms.

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 2 ms.

  Linhas afetadas: 9

d) Table Scan

Para mostrar o comportamento de uma Table Scan, foi criado uma cópia da tabela “CartoesLog” através do comando abaixo:

```
SELECT * INTO CARTOESLOG_SCAN FROM CARTOESLOG
```

Com a inserção, a cópia é apenas dos dados e colunas, os índices não o acompanham, tornando-se uma *heap*.

Após executar o mesmo comando dos exemplos anteriores (SELECT * FROM CARTOESLOG_SCAN WHERE id_pessoafisica = 639224389), o tipo de operação foi Table Scan, fazendo uma varredura, buscando linha por linha da tabela e tendo custo de 100% da operação, além de executar em 3 segundos.

Figura 9 - Table Scan

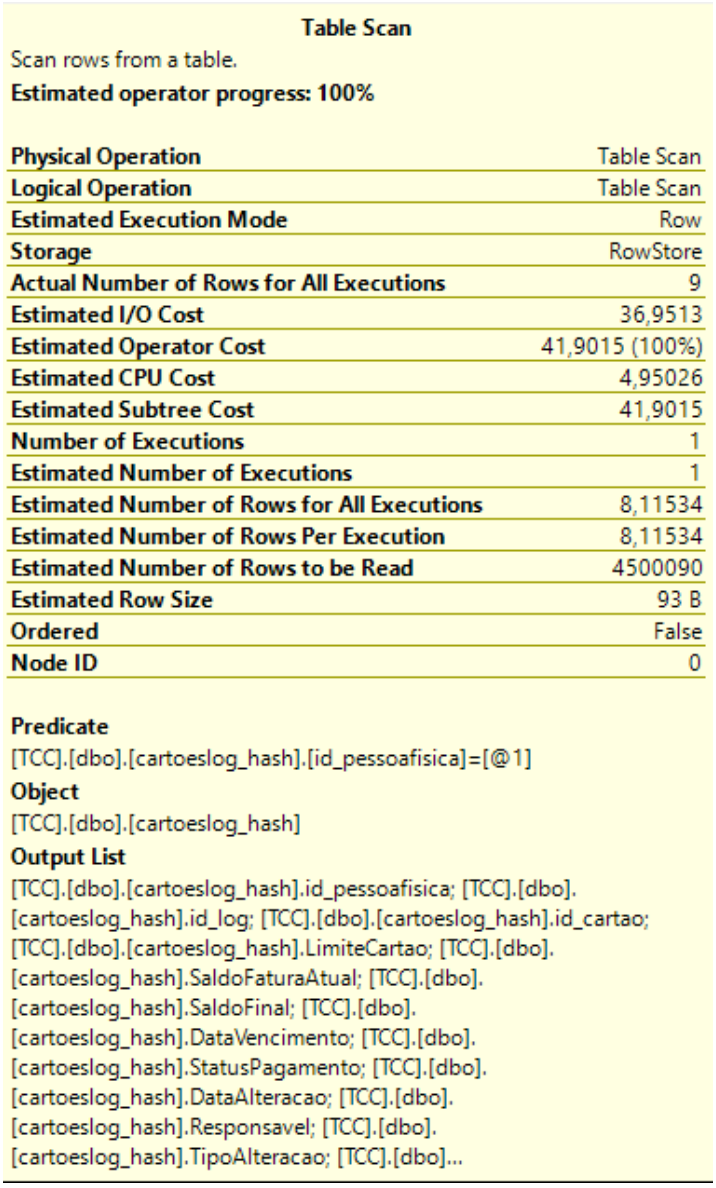


Table Scan

Scan rows from a table.

Estimated operator progress: 100%

Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows for All Executions	9
Estimated I/O Cost	36,9513
Estimated Operator Cost	41,9015 (100%)
Estimated CPU Cost	4,95026
Estimated Subtree Cost	41,9015
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	8,11534
Estimated Number of Rows Per Execution	8,11534
Estimated Number of Rows to be Read	4500090
Estimated Row Size	93 B
Ordered	False
Node ID	0

Predicate
[TCC].[dbo].[cartoeslog_hash].[id_pessoafisica]=[@1]
Object
[TCC].[dbo].[cartoeslog_hash]
Output List
[TCC].[dbo].[cartoeslog_hash].id_pessoafisica; [TCC].[dbo].
[cartoeslog_hash].id_log; [TCC].[dbo].[cartoeslog_hash].id_cartao;
[TCC].[dbo].[cartoeslog_hash].LimiteCartao; [TCC].[dbo].
[cartoeslog_hash].SaldoFaturaAtual; [TCC].[dbo].
[cartoeslog_hash].SaldoFinal; [TCC].[dbo].
[cartoeslog_hash].DataVencimento; [TCC].[dbo].
[cartoeslog_hash].StatusPagamento; [TCC].[dbo].
[cartoeslog_hash].DataAlteracao; [TCC].[dbo].
[cartoeslog_hash].Responsavel; [TCC].[dbo].
[cartoeslog_hash].TipoAlteracao; [TCC].[dbo]...

Fonte: autoria própria.

Resultados obtidos:

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 188 ms, tempo decorrido = 448 ms.

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 875 ms, tempo decorrido = 2798 ms.

Linhas afetadas: 9

5.2 Teste 2: Comandos INNER JOIN vs EXISTS

O segundo teste tem o objetivo de fazer um comparativo ao trazer todos os dados entre duas tabelas, o primeiro usando INNER JOIN e o segundo o comando EXISTS.

a) A primeira consulta traz todos os registros que tem na tabela “Contas” e por consequência, da tabela “CartoesLog”:

```
SELECT c.id_conta, c.SaldoDisponivel, c.id_produto
FROM contas c
INNER JOIN CartoesLOG cc ON cc.id_pessoafisica = c.id_pessoafisica
```

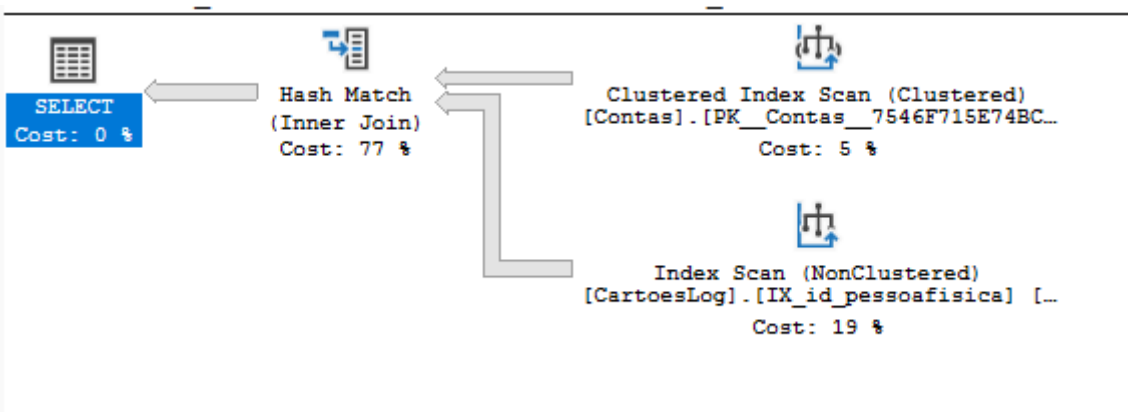
Resultados obtidos:

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 2015 ms, tempo decorrido = 17307 ms.

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 23 ms.
- Tempo de execução em segundos: 17 segundos.
  Linhas afetadas: 5502376

A figura a seguir ilustra o fluxograma de execução. Nela, é possível visualizar que os resultados são unidos através do comando INNER JOIN, gerando maior custo de recursos, mais do que o Index Scan presente.

Figura 10 - Plano de execução da consulta a)



Fonte: autoria própria.

b) A segunda consulta traz todos os dados da tabela “contas”, além de usar uma subconsulta com o comando EXISTS. Dessa forma, o comando verifica se há registros na tabela “cartoes” que tenha um campo “id_pessoafisica” correspondente ao campo “id_pessoafisica” da tabela “contas”.

```
SELECT c.id_conta, c.SaldoDisponivel, c.id_produto
FROM contas c
WHERE EXISTS (
   SELECT 1
   FROM cartoeslog p
   WHERE p.id_pessoafisica = c.id_pessoafisica)
```

Resultados obtidos:

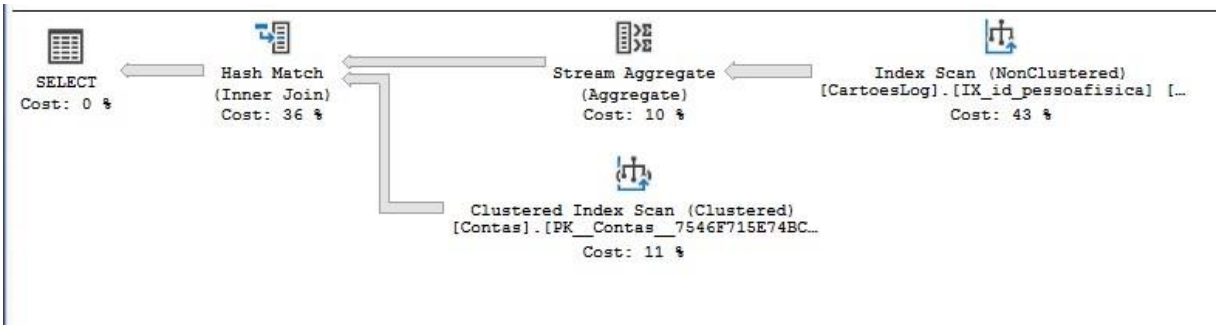
- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 3 ms.

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 2188 ms, tempo decorrido = 7268 ms.
- Tempo de execução em segundos: 04 segundos.

  Linhas afetadas: 500.010

Vejamos, logo abaixo, o plano de execução da segunda consulta:

Figura 11 - Plano de execução da consulta b)



Fonte: autoria própria.

Nesse cenário, a consulta “b” se tornou mais eficiente que a consulta “a”, pois ela para a busca assim que encontra uma correspondência, evitando verificar todos os 5 milhões de registros na tabela “cartoeslog”. Por outro lado, a consulta com INNER JOIN processa todas as combinações possíveis de linhas entre as duas tabelas, o que pode levar mais tempo, especialmente em uma tabela grande.

5.3 Teste 3: Comandos de subtração

O último teste tem o objetivo de comparar consultas com comandos diferentes (EXCEPT, NOT IN, NOT EXISTS) para o mesmo resultado final. Em resumo, o teste irá pegar um registro de uma tabela que não contém em outra tabela, fazendo assim, uma uma operação de subtração.

Nesse caso, foi criado uma tabela chamada “Produtos” que contém a descrição e tipos de produtos financeiros e os respectivos “id_produtos”. Todas as consultas irão procurar um produto que não contenha na tabela “PessoaFisica”, a fim de retornar quais produtos nenhuma conta possui.

a) Operador NOT IN

O operador NOT IN retorna todas as linhas de uma consulta que não possuem uma correspondência em uma lista de valores ou em outra consulta. A consulta executada foi:

```
SELECT id_produto,nome_produto,tipo_produto ,descricao
FROM produtos
WHERE id_produto NOT IN (select id_produto from PessoaFisica)
```

Resultados obtidos:

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 16 ms, tempo decorrido = 24 ms.

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 94 ms, tempo decorrido = 261 ms.

  Linhas afetadas: 1.

b) Operador EXCEPT

O operador EXCEPT retorna todas as linhas de um conjunto de resultados que não estão presentes em outro conjunto de resultados. Consulta a ser executada:

```
SELECT id_produto
FROM produtos
EXCEPT
SELECT id_produto
FROM PessoaFisica
```

Resultados obtidos:

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 0 ms.

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 5 ms.

  Linhas afetadas: 1.

c) Operador NOT EXISTS

Verifica a ausência de correspondência em outra consulta, retornando verdadeiro assim que não encontrar uma linha correspondente.

```
SELECT id_produto,nome_produto,tipo_produto ,descricao
FROM produtos p
WHERE  not exists ( SELECT 1
 FROM PessoaFisica pf
  WHERE pf.id_produto = p.id_produto)
```

Resultados obtidos:

- Tempo de análise e compilação do SQL Server:
  - Tempo de CPU = 0 ms, tempo decorrido = 0 ms.

- Tempos de Execução do SQL Server:
  - Tempo de CPU = 31 ms, tempo decorrido = 67 ms.
    Linhas afetadas: 1.

6. ANÁLISE DOS RESULTADOS

O “Teste 1”, mostra resultados de cenários utilizando tipos de índices, além de realizar consulta sem índice (Teste 1b). Vejamos a tabela:

Tabela 2 - Tempos de execução do Teste 1

	Tempo execução no SQL SERVER (ms)		Tempo análise e compilação (ms)		
	CPU	Tempo decorrido	CPU	Tempo decorrido	Linhas afetadas
Teste 1a Indice Clustered	0	23	0	0	1
Teste 1b Campo sem índice	734	1346	0	22	9
Teste 1c Indice nonClustered	0	2	16	18	9
Teste 1d Table Scan	875	2798	188	448	9

Fonte: autoria própria.

O cenário a1 utilizou o índice clustered em sua PK, como foi consultado com a coluna da própria PK o resultado foi satisfatório. O campo sem índice, “id_pessoafisica”, demonstrou-se uma operação muito custosa para encontrar apenas um registro, porém, ao criar um índice, o tempo decorrido caiu para zero segundos, constatando que uma consulta utilizando índice é mais performático do que uma consulta sem índice. Sendo assim, o Index Seek teve uma execução mais eficiente do que a segunda consulta que utilizou o Index Scan. O pior resultado foi o “Teste d1”, comprovando que o *table scan* possui uma má performance e que deve ser evitado. Os testes foram executados, ao menos, 10 vezes para obter resultados mais precisos.

No “Teste 2”, propomos uma comparação entre INNER JOIN e o comando EXISTS para trazer todos os dados da tabela. Com isso, foi possível perceber melhorias no próprio tempo de execução, como mostra a tabela abaixo:

Tabela 3 - Tempo de execução do Teste 2

	Tempo execução no SQL SERVER (ms)		Tempo análise e compilação (ms)			
	CPU	Tempo decorrido	CPU	Tempo decorrido	Tempo execução (s)	Linhas afetadas
Teste 2a - INNER JOIN	2015	17307	0	23	17	5502376
Teste 2b - EXISTS	2188	7268	0	3	4	500010

Fonte: autoria própria.

No “Teste 2a”, o INNER JOIN trouxe todos os registros de “Contas” e também da “CartoesLog”, junto com as 5 milhões de linhas, pois não teve filtro na consulta. Ou seja, o

comando percorre todas as páginas de dados, apresentando uma lentidão considerável. O ideal da junção entre tabelas é, sempre que possível, filtrar os dados, evitando esse tipo de problema. Sendo assim, a consulta com EXISTS verifica a existência de, pelo menos, uma linha na subconsulta que satisfaça a condição WHERE. Se a subconsulta encontra uma linha, ela retorna "verdadeiro", e a linha da tabela "contas" é incluída nos resultados.

Com isso, o comando EXISTS traz todos os elementos da tabela "contas" sem ter que ler a "CartoesLog" inteira, com uma diferença de 17 segundos para 04 segundos.

Por fim, no "Teste 3", foram feitas comparações entre operadores que trazem os mesmos resultados de subtração. Podemos observar os resultados logo abaixo:

Tabela 4 - Tempo de execução do Teste 3

	Tempo execução no SQL SERVER (ms)		**Tempo análise e compilação (ms)**		
	CPU	**Tempo decorrido**	**CPU**	**Tempo decorrido**	**Linhas afetadas**
Teste 3a - NOT IN	94	621	16	24	1
Teste 3b - EXCEPT	0	5	0	0	1
Teste 3c - NOT EXISTS	31	67	0	0	1

Fonte: autoria própria.

Os resultados foram muito parecidos, não tendo tanta diferença de desempenho nesse teste. Porém, mesmo com resultado sendo apenas um registro, o operador NOT IN demonstrou o maior tempo de execução. Dessa forma, em um cenário de tabelas grandes, pode acontecer do NOT IN não ser o operador indicado para muitos registros.

Por outro lado, o EXCEPT mostrou o melhor resultado. Ele é um operador mais indicado para analisar consultas complexas. Geralmente, o NOT EXISTS tende a ser mais eficiente, principalmente, se tratando de grandes volumes de dados.

Tabela 5 - Comparação de pesquisas relacionadas

Pesquisa Relacionadas	Técnicas de otimização	Resultados obtidos	Cenários utilizados
Gava (2010)	Índices, Planos de Execução, Sintonização (*Tuning*).	Melhoria no desempenho de consultas ao aplicar técnicas de indexação.	Grandes empresas bancárias com grandes volumes de dados.
Rodrigues (2019)	Utilização de Índices em Tabelas, Análise de Planos de Execução.	Análise dos recursos utilizados pelo SQL SERVER através dos planos de execução.	Grandes empresas bancárias com grandes volumes de dados.
Cabral, Corado e Bruno (2015)	Comparação de Operadores e Comandos (INNER JOIN, WHERE, etc.)	Comparação entre consultas com operadores de junção. Resultados de melhorias significativas em termos de tempo de execução.	Grandes empresas bancárias com grandes volumes de dados. Comparação de consultas com diferentes operadores, como junção e subtração.
Gonçalves (2016)	Discussão sobre a utilização de Índices,	Possíveis perdas de desempenho com o uso incorreto de índices.	Discussão teórica com foco nos efeitos colaterais de índices.

Fonte: autoria própria.

Diante do exposto, os objetivos deste trabalho foram atingidos ao analisar e aplicar técnicas de otimização de bancos de dados relacionais em um estudo de caso. A pesquisa de trabalhos relacionados permitiu identificar as principais técnicas de otimização, como índices, planos de execução e sintonização (*tuning*).

Os estudos de Gava (2010), Rodrigues (2019), Cabral, Corado e Bruno (2015) e Gonçalves (2006) forneceram uma base teórica sólida e resultados práticos que destacam a importância e os impactos da otimização de consultas em diferentes cenários.

Foram realizados testes e comparações em cenários que simulam condições reais de produção, mostrando que a aplicação de técnicas de otimização pode melhorar significativamente o desempenho de consultas e reduzir o tempo de resposta em sistemas de gerenciamento de banco de dados.

7. CONCLUSÃO

Este trabalho teve como objetivo principal, analisar técnicas de otimização para bancos de dados relacionais e aplicá-las em um estudo de caso, com o intuito de fazer comparações em cenários de produção num contexto real.

Para isso, foi realizada uma pesquisa abrangente de trabalhos relacionados na área de otimização de bancos de dados, resultando na descrição dos principais métodos de otimização para bancos de dados relacionais, como, indexação, *tuning* e análise de planos de execução.

Os testes elaborados e as comparações realizadas em diferentes cenários, permitiram identificar as melhores práticas e estratégias para otimizar o desempenho dos bancos de dados relacionais. Além disso, os testes contribuíram para o estudo de otimização em banco de dados relacionais, utilizando SQL SERVER.

O estudo também é direcionado ao SGBD, fundamentos de indexação, análise de planos de execuções e comparações entre operadores que podem ser aplicados em outros tipos de SGBDs.

Os resultados obtidos foram satisfatórios. No entanto, os testes foram realizados em uma máquina pessoal de estudo, dificultando na apresentação de muitos registros em grande escala de tempo de execução, em um contexto real com servidores. Contudo, o objetivo de evidenciar a eficácia das técnicas de otimização aplicadas num contexto real, foi alcançado.

Levando em consideração a vasta gama de estudos e pesquisas na área de BDS, acreditamos que o nosso trabalho pode acrescentar na investigação de técnicas de otimização de sistema de banco de dados já existentes e sugerir melhoria de desempenho, como utilização de índices compostos, particionamentos de tabelas, otimização de consultas mais complexas, normalização e modelo relacional, aplicando a performance em um cenário real de grande escala que use concorrências.

REFERÊNCIAS

ALVES, William P. **Banco de Dados**. Editora Saraiva, 2014. E-book. ISBN 9788536518961. Disponível em: < https://integrada.minhabiblioteca.com.br/#/books/9788536518961/ >. Acesso em: 26/04/2024.

BARNHILL, B. **Indexing**. Disponível em: <https://dataschool.com/sql-optimization/how-indexing-works/>. Acesso em: 23/04/2024.

CABRAL, M. K. F., CORADO, V. A., & BRUNO, J. C. (2015, June). **Aplicabilidade de técnicas de otimização de consultas: um estudo de caso**. In 6ª Jice-jornada de iniciação científica e extensão.

CODD, Edgar F. **A relational model of data for large shared data banks**. Communications of the ACM, v. 13, n. 6, p. 377-387, 1970.

CRIVELINI, Wagner. **Chave Primária não é Opcional**. iMasters, 2013 . Disponível em: <https://imasters.com.br/banco-de-dados/chave-primaria-nao-e-opcional>. Acesso em: 23/04/2024.

DATABASES [online]. **Survey Stack Overflow Developer**. Disponível em: < https://survey.stackoverflow.co/2022/#databases>. Acesso em: 20/04/2024.

DATE, C.J. **Introdução a Sistemas de Bancos de Dados**. Grupo GEN, 2004. E-book. ISBN 9788595154322. Disponível em: < https://integrada.minhabiblioteca.com.br/#/books/9788595154322/ >. Acesso em: 26 abr. 2024.

DELANEY, Kalen**. Inside microsoft® sql server™ 2005: query tuning and optimization**. Microsoft Press, 2007. Disponível em: <https://www.devmedia.com.br/metodologia-tuning/11323> Acesso em: 24/04/2024.

DOCUMENTAÇÃO. **Microsoft**, 2022. Disponível em: < https://learn.microsoft.com/pt-br/sql/relational-databases/indexes/heaps-tables-without-clustered-indexes?view=sql-server-ver16 >. Acesso em: 26/04/2024.

ELMASRI, R., & NAVATHE, S. B. (2016). **Fundamentals of Database Systems** (7ª edição). Pearson, 2016.

ELMASRI, R., NAVATHE, S. B., & PINHEIRO, M. G. (2005). **Sistemas de banco de dados**. Pearson Addison Wesley São Paulo. 2005.

FRITCHEY, Grant; DAM, Sajal. **SQL Server Query Performance Tuning**. Apress, 2014.

GAVA, Mariana. **Otimização de consultas em SGBD relacional – estudo de caso**. 2010. Trabalho de Conclusão de Curso - Universidade São Francisco, Itatiba - SP, 2010.

GONÇALVES, E. (2006). **Tuning de SQL: a importância da otimização em comandos SQL**. SQL Magazine, (31), 64-66.

HEUSER, Carlos A. **Projeto de banco de dados - UFRGS**. V.4. . Grupo A, 2011. E-book. ISBN 9788577804528. Disponível em: <

https://integrada.minhabiblioteca.com.br/#/books/9788577804528/ >. Acesso em: 26 abr. 2024.

IKEMATU, Ricardo Shoiti. **Realizando tuning na base de aplicações**. Celepar 2009. Disponível em:< https://www.batebyte.pr.gov.br/Pagina/Realizando-Tuning-na-Base-de-Aplicacoes >. Acesso em: 11/08/2023.

JORGENSEN, Adam et al. **Microsoft SQL server 2012 bible**. John Wiley & Sons, 2012.

MACHADO, Carlos Augusto. **Estudo sobre a otimização de desempenho em banco de dados PostgreSQL**. Repositório de Relatórios-Sistemas de Informação, n. 2, 2014.

MULLINS, Craig S. (2005). **SQL Performance Tuning Basics** [on-line]. Disponível em: < http://www.craigsmullins.com/zjdp_017.htm >. Acesso em: 26/04/2024.

MULLINS, Craig S. **The Most Important Thing is Performance**.1999. Disponível em <http://www.craigsmullins.com/cnr_perf.htm>. Acesso em:11/08/2023.

OLIVEIRA, Kátia Lima de Oliveira; PAIVA, Claudio Eduardo. **Tuning de banco de dados com SQL server**. 2018.

PICINI, Rodrigo; SCHIMIGUEL, Juliano. **Desempenho de Banco de Dados: a Técnica de Otimização com Tuning**. 2008. Disponível em <https://www.devmedia.com.br/metodologia-tuning/11323>. Acesso em: 15/04/2024.

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

RODRIGUES, Nathami Roath. **Um estudo sobre a otimização e melhor performance de consultas no banco de dados sql server**. 2019. Trabalho de Conclusão de Curso - Faculdade Antonio Meneghetti, Restinga Seca - RS, 2019.

SHASHA, Dennis. **Tuning databases for high performance**. ACM Computing Surveys (CSUR), v. 28, n. 1, p. 113-115, 1996.

SILBERSCHATZ, Abraham. **Sistema de Banco de Dados**. Grupo GEN, 2020. E-book. ISBN 9788595157552. Disponível em: < https://integrada.minhabiblioteca.com.br/#/books/9788595157552/> Acesso em: 17/04/2024.

SILBERSCHATZ, Abraham; SUNDARSHAN, S.; KORTH, Henry F. **Sistema de banco de dados**. Elsevier Brasil, 2016.

SILVA, Luiz F C.; RIVA, Aline D.; ROSA, Gabriel A.; et al. **Banco de Dados Não Relacional**. Grupo A, 2021. E-book. ISBN 9786556901534. Disponível em: < https://integrada.minhabiblioteca.com.br/#/books/9786556901534/ >. Acesso em: 17/04/2024.

SORDI, José Osvaldo de. **Modelagem de dados - estudos de casos abrangentes da concepção lógica à implementação**. Editora Saraiva, 2019. *E-book.* ISBN 9788536532370.

Disponível em: < https://integrada.minhabiblioteca.com.br/#/books/9788536532370/ >. Acesso em: 26/04/2024.