DYNAMODB

Eduardo Lucas Lemes Januário

Guilherme Batista de Souza

CATANDUVA

2025

DYNAMODB

"NoSQL não significa 'sem SQL', mas sim 'Não Apenas SQL'"

- Paráfrase ao conceito popularizado por Martin Fowler e Pramod J. Sadalage, em seu livro "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence".

Resumo

A constante evolução das demandas tecnológicas impulsionou uma reavaliação fundamental dos paradigmas de gerenciamento de dados. Em um cenário marcado pela proliferação de informações não estruturadas e semi-estruturadas, bem como pela necessidade imperativa de escalabilidade e performance em tempo real, os sistemas de banco de dados tradicionais, baseados no modelo relacional, revelaram limitações inerentes.

Nesse contexto, emergiu o conceito de bancos de dados NoSQL. Esta abordagem reflete a compreensão de que a diversidade de desafios computacionais exige um portfólio igualmente diverso de ferramentas de persistência de dados. Tais sistemas transcendem a rigidez das tabelas e esquemas fixos, oferecendo modelos de dados alternativos, como pares chave-valor, documentos, colunas amplas ou grafos. Essa flexibilidade intrínseca se alinha perfeitamente com a natureza dinâmica dos dados modernos, proporcionando uma capacidade de organização mais adaptável às necessidades específicas de aplicações que operam com vastos volumes de informações e exigem alta vazão de operações de leitura e escrita.

Este estudo explorará as características distintivas do Amazon DynamoDB, um serviço NoSQL que oferece performance e escalabilidade em ambientes de alta demanda, e seu impacto em aplicações de grande porte.

Abstract

The continuous evolution of technological demands has prompted a fundamental reevaluation of data management paradigms. In a landscape characterized by the proliferation of unstructured and semi-structured information, alongside the imperative need for real-time scalability and performance, traditional database systems, based on the relational model, have revealed inherent limitations.

In this context, the concept of NoSQL databases emerged. This approach reflects the understanding that the diversity of computational challenges necessitates an equally diverse portfolio of data persistence tools. Such systems transcend the rigidity of fixed tables and schemas, offering alternative data models like key-value pairs, documents, wide columns, or graphs. This intrinsic flexibility aligns perfectly with the dynamic nature of modern data, providing an organizational capacity more adaptable to the specific needs of applications operating with vast volumes of information and requiring high throughput for read and write operations.

This study will explore the distinctive characteristics of Amazon DynamoDB, a NoSQL service that delivers performance and scalability in high-demand environments, and its impact on large-scale applications.

Índice

1.Introdução

Um banco de dados não relacional é um sistema de armazenamento de dados que não utiliza o modelo tradicional de tabelas compostas por linhas e colunas, como nos bancos de dados relacionais. Em vez disso, esses bancos empregam modelos de dados alternativos e mais flexíveis, como estruturas de pares chave-valor, documentos no formato JSON, colunas amplas ou grafos baseados em vértices e arestas. Essa abordagem permite que os dados sejam organizados de maneira mais dinâmica, adequada às necessidades específicas de determinadas aplicações, especialmente aquelas que lidam com grandes volumes de informações, estrutura de dados variável e que demandam alta performance em operações massivas de leitura e escrita.

Os bancos de dados NoSQL (*Not Only SQL*) surgiram com o objetivo de superar limitações comuns aos sistemas relacionais tradicionais, principalmente no que se refere à escalabilidade horizontal, à flexibilidade no armazenamento e à eficiência sob cargas de trabalho intensas e dinâmicas. Tornaram-se fundamentais em ambientes computacionais modernos, como aplicações web em tempo real, plataformas de mídia social, serviços de streaming, sistemas de recomendação, Internet das Coisas (IoT) e análise de dados em larga escala. Cada modelo NoSQL é projetado para atender a diferentes exigências, oferecendo características específicas de desempenho, estruturação e organização de dados.

Atualmente, os bancos NoSQL são classificados em quatro grandes categorias, de acordo com sua modelagem de dados. O primeiro modelo é o chave-valor, no qual os dados são armazenados como pares compostos por uma chave única e um valor, permitindo acesso extremamente rápido. O segundo modelo é o de documentos, que organiza os dados em estruturas semelhantes a documentos JSON ou BSON, oferecendo flexibilidade e escalabilidade, permitindo que diferentes registros tenham estruturas distintas. O terceiro modelo é o colunar, no qual os dados são armazenados por colunas em vez de linhas, sendo altamente eficiente para consultas analíticas, agregações e grandes volumes de dados transacionais. Por fim, há o modelo de grafos, voltado para o armazenamento de dados altamente inter-relacionados, sendo ideal para redes sociais, sistemas de roteamento e motores de recomendação.

No desenvolvimento deste trabalho, o foco recaiu sobre o banco de dados Amazon DynamoDB, uma solução NoSQL gerenciada e oferecida pela AWS, que combina as arquiteturas chave-valor e documento. O DynamoDB destaca-se por sua alta escalabilidade, baixa latência, replicação global e modelo de cobrança baseado em demanda, sendo amplamente utilizado em aplicações que exigem desempenho constante e operações em grande escala. O projeto desenvolvido tem como objetivo compreender e aplicar, na prática, os conceitos do DynamoDB, demonstrando seu funcionamento por meio da construção de um sistema real de controle de estoque, denominado Turtle Track, que utiliza essa tecnologia como backend para o gerenciamento de dados.

Esse projeto não se limita apenas à exploração teórica dos conceitos de bancos NoSQL, mas também propõe a aplicação prática desses conhecimentos, construindo uma solução funcional e aplicável ao mercado. Dessa forma, busca-se não apenas compreender as vantagens e limitações do DynamoDB, mas também avaliar, na prática, sua eficiência, escalabilidade, robustez e aplicabilidade em um cenário concreto de desenvolvimento de software voltado ao gerenciamento de estoques.

2.DynamoDB

O Amazon DynamoDB é um banco de dados não relacional desenvolvido e disponibilizado pela Amazon Web Services (AWS), cujo projeto visa oferecer uma solução altamente escalável, distribuída e de baixa latência, adequada às demandas das aplicações modernas que operam em larga escala. Ao longo dos anos, consolidou-se como uma das soluções NoSQL mais robustas do mercado, sendo amplamente utilizado por empresas de diferentes setores que necessitam de alta disponibilidade, performance consistente e elasticidade operacional.

2.1 Histórico e Desenvolvedor Principal

O desenvolvimento do DynamoDB tem origem direta nas necessidades internas da própria Amazon, que, no início dos anos 2000, enfrentava dificuldades para garantir alta disponibilidade, tolerância a falhas e escalabilidade em seus sistemas de backend, especialmente durante eventos sazonais como a Black Friday, quando havia picos extremos de acesso.

Em 2007, engenheiros da Amazon publicaram o artigo técnico intitulado “Dynamo: Amazon's Highly Available Key-Value Store”, que descrevia um sistema interno de banco de dados distribuído, projetado para garantir alta disponibilidade, mesmo em cenários de falhas de hardware e redes. Esse documento se tornou um marco na história dos bancos NoSQL, influenciando significativamente a indústria.

Baseando-se nos princípios descritos nesse artigo — como particionamento de dados, replicação, consistência eventual e resiliência a falhas —, a Amazon lançou oficialmente, em janeiro de 2012, o Amazon DynamoDB, que representa a evolução do conceito original. Ao contrário do Dynamo interno, que demandava manutenção manual e configuração complexa, o DynamoDB foi projetado como um serviço totalmente gerenciado e serverless, eliminando a necessidade de administração da infraestrutura por parte dos usuários. Isso inclui escalabilidade automática, gerenciamento de nós, balanceamento de carga e provisionamento de capacidade, tornando-se altamente atraente tanto para startups quanto para grandes corporações.

2.2 Modelo de Dados Utilizado

O DynamoDB adota um modelo de dados que combina os paradigmas chave-valor e documento, oferecendo grande flexibilidade na organização e no armazenamento das informações. Os dados são estruturados em tabelas, mas, diferentemente dos bancos de dados relacionais, não exigem esquemas fixos. Isso permite que cada item (ou registro) dentro da mesma tabela possua um conjunto de atributos distinto, com diferentes tipos e estruturas.

Cada item é obrigatoriamente identificado por uma chave primária, que pode ser composta de duas formas:

- Chave simples, utilizando apenas um atributo denominado chave de partição (partition key).
- Chave composta, formada pela combinação de uma chave de partição e uma chave de ordenação (sort key), o que permite armazenar múltiplos itens que compartilham a mesma chave de partição, diferenciando-os pela sort key.

Além disso, o DynamoDB permite a criação de Índices Secundários Globais (GSI) e Índices Secundários Locais (LSI), que possibilitam consultas adicionais sobre atributos que não fazem parte da chave primária, aumentando significativamente a flexibilidade na realização de buscas e ordenações complexas.

Por trabalhar com um modelo sem esquema rígido, o DynamoDB é altamente adaptável a mudanças nos requisitos das aplicações, permitindo adicionar ou remover atributos dos itens sem a necessidade de alterar a estrutura da tabela, o que oferece agilidade no desenvolvimento e manutenção de sistemas.

2.3 Linguagem de Consulta

Ao contrário dos bancos de dados relacionais, que utilizam SQL (Structured Query Language) como linguagem padrão de consulta, o DynamoDB opera por meio de uma interface baseada em chamadas de API, acessíveis através dos SDKs oficiais da AWS, como o Boto3 para Python, o AWS SDK para Java, entre outros. As operações de leitura, escrita, atualização e exclusão são executadas diretamente por essas APIs, utilizando dados estruturados no formato JSON.

Para consultas mais elaboradas, o DynamoDB oferece suporte à linguagem PartiQL, uma extensão declarativa que permite realizar operações semelhantes às do SQL em estruturas NoSQL, facilitando a curva de aprendizado para desenvolvedores acostumados com bancos relacionais. Mesmo assim, PartiQL ainda possui limitações em relação a junções (*joins*) e operações relacionais complexas.

O banco oferece dois modos principais de leitura:

- Leitura consistente eventual, que prioriza desempenho e escalabilidade, permitindo que as leituras possam, em alguns casos, não refletir imediatamente a última atualização.

- Leitura fortemente consistente, que garante que a leitura sempre retorne a versão mais atual dos dados, embora com maior consumo de throughput e, potencialmente, maior latência.

O DynamoDB também suporta operações de escaneamento completo da tabela (Scan) e consultas filtradas por chave (Query), sendo este último muito mais eficiente, pois opera diretamente sobre os índices primários ou secundários.

2.4 Casos de uso e aplicações reais

O DynamoDB é utilizado por algumas das maiores e mais exigentes empresas do mundo, cuja operação depende diretamente de bancos de dados com alta disponibilidade, latência ultrabaixa e capacidade de escalar horizontalmente sem comprometimento da performance.

Um exemplo notável é a Snap Inc., responsável pelo aplicativo Snapchat, que adotou o DynamoDB como parte central de sua arquitetura distribuída baseada em microserviços. A migração de uma infraestrutura monolítica para uma estrutura distribuída com DynamoDB resultou em uma redução superior a 20% na latência mediana para envio de mensagens, além de permitir que o sistema processe mais de 10 milhões de requisições por segundo durante picos de uso. A adoção dessa arquitetura também contribuiu para uma redução expressiva nos custos operacionais relacionados à infraestrutura.

No segmento de streaming de mídia, empresas como Netflix, Hulu e Disney+ utilizam o DynamoDB para gerenciar dados críticos, como preferências de usuários, sessões ativas, listas personalizadas, catálogos de conteúdo e controle de dispositivos, garantindo uma experiência de usuário fluida mesmo durante picos massivos de audiência, como lançamentos de filmes ou séries populares.

No setor financeiro, bancos e fintechs empregam o DynamoDB para processamento de transações em tempo real, gerenciamento de carteiras digitais, históricos de clientes e monitoramento antifraude. A capacidade do DynamoDB de replicar dados entre múltiplas regiões com latência inferior a milissegundos é essencial para garantir a segurança, a consistência e a disponibilidade contínua dos serviços.

O DynamoDB também é amplamente adotado em setores como e-commerce, jogos online, IoT (Internet das Coisas) e inteligência artificial, evidenciando sua versatilidade para aplicações que exigem alta escalabilidade, disponibilidade global, consistência e performance previsível.

Estes casos demonstram, de forma concreta, que o DynamoDB não é apenas uma solução para grandes corporações, mas também um recurso acessível e escalável para startups, pequenas e médias empresas que precisam de uma infraestrutura de banco de dados confiável, elástica e de fácil manutenção, adaptando-se perfeitamente às exigências da transformação digital atual.

3.Comparativo

Os bancos de dados relacionais (SQL) e os bancos de dados não relacionais (NoSQL) adotam filosofias fundamentalmente distintas quanto ao armazenamento e gerenciamento de dados. Enquanto os bancos relacionais, como MySQL, PostgreSQL e Oracle, utilizam uma estrutura rígida baseada em tabelas com esquemas predefinidos, os bancos NoSQL como DynamoDB, MongoDB e Cassandra oferecem uma abordagem mais flexível, geralmente sem esquemas fixos e com suporte a formatos variados como chave-valor, documentos JSON, colunas ou grafos.

NoSQL surgiu da necessidade de lidar com grandes volumes de dados não estruturados, escalabilidade horizontal e alta disponibilidade, especialmente em aplicações modernas como redes sociais, IoT e comércio eletrônico. Por outro lado, bancos SQL são altamente eficazes em garantir integridade e consistência de dados por meio de transações ACID — essenciais em aplicações financeiras, por exemplo.

As diferenças também se refletem na forma de escalar os sistemas. Enquanto os bancos SQL geralmente escalam verticalmente (aumentando a capacidade de um único servidor), os bancos NoSQL são projetados para escalar horizontalmente (adicionando múltiplos servidores). Além disso, os relacionais usam SQL como linguagem padrão de consulta, enquanto os bancos NoSQL frequentemente utilizam APIs próprias ou linguagens de consulta mais específicas, como o PartiQL no DynamoDB.

Critério	**Banco de Dados Relacional (SQL)**	**Banco de Dados Não Relacional (NoSQL)**
Modelo de dados	Tabelas com esquema fixo	Flexível (chave-valor, documento, etc.)
Linguagem de consulta	SQL	API própria, JSON, PartiQL, etc.
Consistência	Forte (ACID)	Eventual (BASE), com exceções
Escalabilidade	Vertical	Horizontal

Tipos de dados	Estruturados	Semi ou não estruturados
Ideal para...	Transações complexas, integridade	Grandes volumes de dados, alta velocidade
Gerenciamento	Manual ou com automação parcial	Totalmente gerenciado (em muitos casos)
Exemplo	MySQL, PostgreSQL, Oracle	DynamoDB, MongoDB, Cassandra

Ao comparar o DynamoDB com outro banco de dados NoSQL do tipo documento, o MongoDB surge como uma escolha natural. Ambos suportam dados no formato JSON, são altamente escaláveis e são amplamente utilizados por grandes empresas para aplicações modernas. No entanto, suas arquiteturas e formas de operação diferem significativamente.

O DynamoDB, oferecido como serviço totalmente gerenciado pela AWS, é focado em desempenho extremo, com escalabilidade automática, alta disponibilidade e baixa latência. Ele combina os modelos chave-valor e documento, mas com forte ênfase em throughput e infraestrutura "serverless". Já o MongoDB, de código aberto, é mais flexível no uso de documentos aninhados e operações complexas de agregação, além de permitir instalação local (on-premises) ou em nuvens de múltiplos provedores.

Uma das maiores diferenças está no modelo de gerenciamento: enquanto o DynamoDB elimina completamente a necessidade de administrar servidores ou ajustar performance, o MongoDB exige configuração e manutenção contínua — a menos que seja usado com o serviço gerenciado MongoDB Atlas. Além disso, o DynamoDB é altamente integrado ao ecossistema AWS, oferecendo segurança, controle de acesso e replicação regional com um clique, o que o torna ideal para quem já utiliza serviços da Amazon.

A escolha pelo DynamoDB se justifica principalmente pela sua robustez e simplicidade operacional. Para aplicações que exigem baixa latência em escala global, o DynamoDB proporciona um desempenho previsível e confiável, sem a necessidade de gerenciamento de infraestrutura. Além disso, sua integração nativa com outros serviços da AWS (como Lambda, S3, CloudWatch, etc.) torna-o uma excelente opção para arquiteturas modernas e orientadas a eventos. Apesar do MongoDB oferecer maior flexibilidade na modelagem de dados, o DynamoDB leva vantagem em cenários onde performance, escalabilidade e simplicidade de manutenção são essenciais.

Critério	**DynamoDB (AWS)**	**MongoDB**
Tipo	Chave-valor + Documento JSON	Documento JSON

Gerenciamento	Totalmente gerenciado (serverless)	Local ou gerenciado (Atlas)
Linguagem de consulta	APIs AWS + PartiQL	MongoDB Query Language (MQL)
Operações de agregação	Limitadas	Muito poderosas (aggregation pipeline)
Escalabilidade	Automática e horizontal	Manual (sharding) ou automatizada (Atlas)
Integração com cloud	Nativa com AWS	Multicloud (Azure, GCP, AWS via Atlas)
Performance	Otimizado para leitura/gravação de alta velocidade	Equilibrado, com foco em modelagem flexível
Modelo de consistência	Eventual (com suporte a forte em leitura)	Forte (com consistência configurável)
Backup e segurança	Nativos e automatizados	Necessita configuração (ou Atlas)

Ao comparar bancos de dados relacionais e não relacionais, fica evidente que cada paradigma atende a necessidades distintas do mundo tecnológico atual. Os bancos relacionais, como o MySQL, continuam sendo essenciais para aplicações que demandam integridade transacional, estrutura rígida e relacionamentos complexos entre dados. Já os bancos NoSQL, como o DynamoDB, se destacam pela flexibilidade, escalabilidade horizontal e desempenho em tempo real, sendo altamente eficazes em cenários modernos de alto volume e variabilidade de dados.

Especificamente no universo dos bancos de dados não relacionais, o DynamoDB demonstra uma abordagem robusta e otimizada para aplicações em escala, como redes sociais, e-commerce e sistemas com grande número de leituras e gravações simultâneas. Quando comparado ao MongoDB, por exemplo, o DynamoDB oferece simplicidade operacional e integração profunda com a nuvem AWS, características fundamentais para projetos com foco em desempenho e mínima administração de infraestrutura.

No entanto, apesar de suas vantagens, o DynamoDB também apresenta limitações — como suporte limitado a operações de agregação e dependência do ecossistema AWS — que precisam ser consideradas no momento da escolha tecnológica. Assim, torna-se essencial analisar cuidadosamente

os prós e contras dessa solução para decidir se ela realmente atende às necessidades específicas de cada projeto.

4.Prós e Contras

O Amazon DynamoDB é amplamente reconhecido como uma solução de banco de dados não relacional que atende às exigências de aplicações modernas que demandam alta escalabilidade, desempenho consistente e disponibilidade global. Projetado para ambientes de grande volume e com intensa variação de carga, esse banco de dados oferece uma infraestrutura gerenciada e automatizada, com foco na eliminação de tarefas administrativas relacionadas à manutenção de servidores, replicação de dados e gerenciamento de desempenho. Sua integração nativa ao ecossistema da Amazon Web Services (AWS) o torna ainda mais atrativo para organizações que já operam em nuvem ou pretendem migrar seus sistemas para ambientes altamente escaláveis.

No entanto, apesar de suas vantagens evidentes, o DynamoDB apresenta limitações que devem ser cuidadosamente analisadas antes de sua adoção. Um dos principais desafios enfrentados por equipes de desenvolvimento é a necessidade de adaptação a um novo modelo de dados, que difere substancialmente do paradigma relacional tradicional. A ausência de suporte nativo a operações como junções entre tabelas, consultas complexas e transações relacionais exige uma mudança na lógica de modelagem e consulta dos dados. Adicionalmente, embora a linguagem PartiQL ofereça uma interface semelhante ao SQL, suas funcionalidades permanecem limitadas em relação aos bancos de dados relacionais convencionais.

Experiências práticas de grandes corporações que adotaram o DynamoDB reforçam seu potencial. A Snap Inc., desenvolvedora do Snapchat, obteve significativa melhoria de desempenho após a adoção do serviço, alcançando mais de dez milhões de requisições por segundo em momentos de pico e reduzindo em mais de 20% a latência de envio de mensagens. A Netflix, por sua vez, utiliza o DynamoDB para armazenar preferências de usuários, histórico de exibição e dados de personalização de conteúdo, assegurando um serviço ágil e escalável mesmo em horários de alta audiência. O próprio marketplace da Amazon integra o DynamoDB em componentes críticos como carrinho de compras, gerenciamento de sessões e rastreamento de pedidos, aproveitando sua integração total com os demais serviços da nuvem AWS.

Apesar desses resultados positivos, há registros de dificuldades enfrentadas por desenvolvedores e administradores ao lidar com o custo operacional da ferramenta. Em muitos casos, a má configuração de índices secundários, uso ineficiente de leituras consistentes e ausência de otimização no particionamento de dados podem resultar em aumentos significativos nos custos mensais. A cobrança baseada em throughput provisionado ou no consumo sob demanda exige atenção constante, planejamento adequado e monitoramento frequente, especialmente em ambientes com variação imprevisível de carga.

Categoria	Prós	Contras
Desempenho	Baixa latência mesmo sob carga intensa	Requisições consistentes podem ser mais lentas e custosas
Escalabilidade	Escalabilidade horizontal automática, sem intervenção manual	Gargalos podem ocorrer se o particionamento for mal estruturado
Gerenciamento	Totalmente gerenciado, reduzindo a sobrecarga operacional	Dependência do ecossistema AWS

Modelo de dados	Flexível, sem necessidade de esquema fixo	Exige nova abordagem de modelagem, diferente do paradigma relacional
Segurança e backup	Suporte nativo a backup automático, replicação regional e criptografia	Configurações de segurança mais avançadas requerem conhecimento técnico
Linguagem de consulta	APIs bem documentadas e suporte a PartiQL	Funcionalidades limitadas para consultas complexas e agregações
Custos	Modelo de cobrança baseado no uso real, com modo on-demand disponível	Custos imprevisíveis em cenários de tráfego irregular ou mal dimensionado
Casos de uso	Utilizado por grandes empresas, como Snapchat, Netflix e Amazon	Pouco indicado para aplicações com lógica de negócio altamente relacional

Considerando todos os fatores analisados, verifica-se que as vantagens do DynamoDB superam suas limitações, especialmente quando utilizado em contextos que demandam alta performance, disponibilidade contínua e escalabilidade dinâmica. Em projetos com arquitetura moderna, baseados em microsserviços, eventos em tempo real ou serviços globais, o DynamoDB apresenta-se como uma escolha técnica sólida e eficiente. Entretanto, sua adoção deve ser precedida de uma avaliação criteriosa do perfil do sistema e da equipe técnica, uma vez que sua utilização inadequada pode acarretar custos operacionais elevados e dificuldades de manutenção. Em síntese, o DynamoDB é altamente recomendável para soluções orientadas a volume e desempenho, desde que seu uso esteja alinhado com as características do projeto e com boas práticas de modelagem e operação em bancos NoSQL.

5.Aplicação Desenvolvida - TurtleTrack

5.1 Introdução

O Turtle Track surge como uma solução eficiente para o controle de estoque de empresas, negócios locais e pequenos empreendimentos que necessitam de um gerenciamento prático, seguro e acessível. Seu desenvolvimento foi pensado para atender à necessidade de controle preciso sobre produtos, oferecendo funcionalidades que permitem desde o cadastro, busca, atualização até a exclusão de itens, tudo por meio de uma interface gráfica intuitiva e de fácil utilização. A proposta central do sistema está fundamentada na combinação de uma interface amigável com uma infraestrutura robusta, baseada na nuvem, capaz de proporcionar aos usuários uma gestão simplificada, porém altamente eficaz, de seus estoques.

A crescente demanda por soluções tecnológicas que viabilizem o gerenciamento eficiente de estoques reflete a transformação digital pela qual as empresas vêm passando nas últimas décadas. Cada vez mais, organizações buscam ferramentas que combinem praticidade, escalabilidade, segurança e disponibilidade, a fim de substituir métodos manuais e planilhas, que são suscetíveis a erros, inconsistências e perda de dados. Nesse cenário, o Turtle Track se posiciona como uma resposta tecnológica moderna, capaz de suprir essas carências operacionais. A aplicação foi desenvolvida utilizando a linguagem de programação Python, integrando-se diretamente ao serviço de banco de dados não relacional Amazon DynamoDB, que oferece alta disponibilidade, escalabilidade automática e performance consistente, características essenciais para sistemas que demandam confiabilidade e operação contínua.

Este projeto reflete não apenas uma solução funcional para controle de estoque, mas também representa uma aplicação prática de conceitos atuais de desenvolvimento de software e arquitetura de dados, unindo programação, experiência de usuário e tecnologia em nuvem. Por meio da adoção do DynamoDB, o sistema elimina limitações comuns dos bancos de dados tradicionais, oferecendo maior flexibilidade na modelagem dos dados e garantindo desempenho superior mesmo em cenários de alta demanda. Dessa forma, o Turtle Track se apresenta como uma ferramenta capaz de potencializar a organização, o controle e a eficiência operacional dos estoques, alinhando-se às necessidades das empresas na era da transformação digital.

5.2 Contextualização e Problematica

No ambiente empresarial atual, a gestão de estoque é um dos pilares da sustentabilidade operacional. Erros manuais, perda de informações, falta de histórico e dificuldade na visualização dos dados são problemas recorrentes em empresas que ainda realizam o controle de seus estoques de forma manual ou com ferramentas não especializadas, como planilhas. Esses métodos são propensos a falhas, não oferecem atualizações em tempo real e dificultam o acompanhamento da disponibilidade dos produtos.

Além disso, sistemas tradicionais baseados em bancos de dados relacionais podem se tornar caros, pouco escaláveis e complexos de gerenciar para empresas que buscam simplicidade e flexibilidade. Surge, portanto, a necessidade de uma solução que combine um backend eficiente e escalável, como o DynamoDB, com uma aplicação desktop de fácil utilização, capaz de atender as principais demandas operacionais de controle de estoque.

5.3 Descrição Funcional da Aplicação

O Turtle Track foi desenvolvido para atender às principais necessidades operacionais no gerenciamento de estoque, oferecendo um conjunto robusto de funcionalidades que permitem controle, organização e manutenção eficiente dos produtos. A seguir, são descritas detalhadamente as principais operações contempladas pelo sistema.

- Cadastro de Produtos

  O cadastro de produtos é uma das funções centrais do Turtle Track e tem como objetivo garantir que todos os itens do estoque estejam devidamente registrados, organizados e identificados de maneira única e precisa. Por meio dessa funcionalidade, o usuário insere informações fundamentais, como o código identificador (ID) do produto, nome, quantidade disponível em estoque e preço unitário.

  Esse processo é essencial não apenas para a organização do estoque, mas também para assegurar a integridade e a rastreabilidade dos dados. Cada produto é registrado utilizando uma chave única, denominada product_id, que funciona como a chave primária na tabela do DynamoDB. Isso evita duplicidade de registros e permite que todas as operações futuras, como consultas, atualizações e deleções, sejam realizadas de forma eficiente e segura.

  Além disso, a aplicação realiza uma verificação automática para garantir que não existam produtos com o mesmo ID ou nome, prevenindo erros e inconsistências no banco de dados. O cadastro bem estruturado reflete diretamente na qualidade do controle de estoque e na precisão das informações acessadas pelos usuários.

- Busca e Exibição de Produtos

  A funcionalidade de busca permite ao usuário consultar rapidamente qualquer

produto armazenado no banco de dados, utilizando como critério principal o código identificador (ID) do item. Esse mecanismo foi projetado para oferecer respostas rápidas e precisas, permitindo que o operador visualize informações como nome, quantidade em estoque e preço do produto em tempo real.

Essa operação é fundamental para auxiliar processos logísticos e operacionais, como conferências de estoque, atendimento a clientes, verificações de disponibilidade e auditorias internas. A agilidade proporcionada pela integração com o DynamoDB, que possui baixíssima latência na recuperação de dados, garante uma experiência de consulta extremamente eficiente.

A interface da busca foi desenhada para ser intuitiva, onde o usuário insere o código do produto e recebe imediatamente os dados associados, apresentados de maneira clara e organizada. Essa funcionalidade também serve como ponto de partida para operações subsequentes, como edição e exclusão de produtos.

- Edição e Atualização de Produtos

A edição de produtos é uma funcionalidade indispensável em qualquer sistema de controle de estoque, visto que os dados dos produtos estão sujeitos a alterações frequentes, seja por mudanças no preço de venda, ajustes nas quantidades após inventários físicos, correções de cadastro ou modificações na nomenclatura dos itens.

O Turtle Track permite que o usuário selecione um produto existente e realize atualizações nos campos de nome, quantidade e preço. O sistema garante que, antes de qualquer modificação, seja realizada uma validação para evitar conflitos, como o uso de um nome já atribuído a outro produto.

Essa funcionalidade garante não apenas a consistência dos dados, mas também a sua relevância e atualidade, permitindo que o estoque reflita, com precisão, a realidade operacional da empresa. Além disso, todo o processo de edição é realizado de forma simples, segura e eficiente, utilizando comandos de atualização direta no DynamoDB, que respondem quase que instantaneamente às mudanças solicitadas.

- Exclusão de Produtos

A remoção de produtos do sistema é uma função essencial para manter o banco de dados limpo, organizado e livre de registros desnecessários ou obsoletos. Essa operação é utilizada principalmente em casos de itens descontinuados, produtos com cadastro duplicado ou erros que necessitem a exclusão definitiva do registro.

Ao selecionar um produto para exclusão, o sistema executa uma operação segura no DynamoDB, utilizando o product_id como chave para localizar e remover o item da base de dados. Antes da execução da exclusão, é possível revisar as informações do produto, minimizando riscos de remoções equivocadas.

Esse controle permite que o histórico de produtos permaneça atualizado, evitando que itens descontinuados interfiram em processos de gestão, relatórios de estoque ou consultas operacionais. A prática de manutenção periódica do banco, incluindo a exclusão de registros inválidos, é essencial para a saúde do sistema e para garantir a performance otimizada do banco de dados.

A combinação dessas quatro funcionalidades — cadastro, busca, edição e exclusão — oferece uma solução robusta, escalável e eficiente para o gerenciamento de estoques. O Turtle Track, portanto, não apenas automatiza processos operacionais, como também assegura a

integridade dos dados, reduz erros manuais e oferece uma experiência de uso altamente confiável, alinhada às melhores práticas de controle logístico e gestão de inventário.

5.4 Arquitetura e Fundamentos Técnicos

A arquitetura do Turtle Track adota uma abordagem moderna, fundamentada no paradigma cliente-nuvem, na qual a aplicação opera localmente por meio de uma interface gráfica desenvolvida em Python, enquanto o armazenamento e a persistência dos dados ocorrem na nuvem, utilizando o serviço gerenciado Amazon DynamoDB. Esse modelo arquitetônico proporciona uma combinação eficiente entre simplicidade de uso no front-end e robustez, escalabilidade e resiliência no back-end.

A camada de interface foi construída com a biblioteca Tkinter, que oferece os recursos necessários para o desenvolvimento de interfaces gráficas desktop em Python, incluindo a criação de janelas, botões, caixas de texto, menus e navegação entre diferentes abas funcionais. Essa abordagem garante uma experiência de usuário intuitiva, permitindo que operadores sem formação técnica consigam manusear o sistema com facilidade e segurança.

No backend, o DynamoDB, serviço NoSQL da AWS, foi escolhido por suas características superiores em termos de desempenho, disponibilidade e escalabilidade horizontal. Trata-se de um banco de dados orientado a chave-valor e documentos, que não requer a definição de esquemas rígidos como nos bancos de dados relacionais. Isso permite que diferentes itens na mesma tabela possuam atributos distintos, tornando o modelo de dados altamente flexível e adaptável às mudanças, sem a necessidade de alterações estruturais na base.

O DynamoDB opera por meio de uma chave primária, que no caso da aplicação é representada pelo atributo product_id. Essa chave garante a unicidade dos registros e permite que as operações de leitura, escrita, atualização e exclusão sejam extremamente rápidas, mesmo em tabelas com milhões de itens. Além disso, o DynamoDB oferece suporte à criação de Índices Secundários Globais (GSI) e Índices Secundários Locais (LSI), que possibilitam consultas eficientes por outros atributos além da chave primária, viabilizando pesquisas dinâmicas e filtragens mais sofisticadas quando necessário.

A escolha do DynamoDB se fundamenta também em sua capacidade de escalabilidade automática, que permite que o sistema absorva aumentos repentinos na carga de trabalho, seja em momentos de crescimento do volume de dados ou de picos de acesso simultâneo, sem degradação perceptível de desempenho. Essa escalabilidade é alcançada graças à arquitetura distribuída do banco, que divide os dados em múltiplas partições e replica automaticamente essas partições entre diferentes servidores e zonas de disponibilidade dentro da infraestrutura da AWS.

Adicionalmente, o DynamoDB oferece recursos como provisionamento de throughput, tanto em modo sob demanda quanto em capacidade provisionada, mecanismos de replicação global (Global Tables), que garantem baixa latência de acesso a partir de diferentes regiões geográficas, e suporte a backup e restauração automáticos, essenciais para garantir a resiliência e a continuidade do negócio.

Diferentemente dos bancos de dados relacionais tradicionais, que operam com modelos baseados em tabelas com esquemas fixos, integridade referencial e dependência de normalização, o DynamoDB permite um modelo mais flexível, orientado a entidades independentes. Isso elimina a necessidade de operações complexas como *joins*, que são custosas em termos de processamento, e favorece uma modelagem otimizada para consultas rápidas e operações simples de leitura e escrita, altamente adequadas para sistemas de controle de estoque.

Esse modelo de dados simplificado e flexível, combinado com a resiliência da arquitetura distribuída da AWS, garante que o Turtle Track opere de forma estável, rápida e escalável, atendendo tanto pequenas operações quanto cenários de crescimento acelerado, sem que sejam necessárias

intervenções manuais na infraestrutura.

Dessa forma, a arquitetura do sistema não apenas atende aos requisitos funcionais do controle de estoque, como também oferece uma base tecnológica robusta, alinhada às melhores práticas de desenvolvimento de software na nuvem e gestão de dados na era da transformação digital.

5.5 Integração Pratica e Implementação da Aplicação

A implementação da aplicação Turtle Track foi concebida para garantir eficiência, simplicidade operacional e robustez tecnológica, aliando uma interface gráfica intuitiva com uma infraestrutura de dados altamente escalável. Toda a integração da aplicação com o serviço de banco de dados na nuvem Amazon DynamoDB foi realizada utilizando a biblioteca Boto3, o SDK oficial da AWS para a linguagem Python, que oferece suporte completo às operações de manipulação de dados na nuvem.

Essa integração permite que o sistema execute, de forma direta e eficiente, todas as operações essenciais de banco de dados, conhecidas pela sigla CRUD (Create, Read, Update e Delete). O banco de dados foi estruturado com uma tabela principal, na qual cada produto é armazenado utilizando o campo product_id como chave primária. Essa configuração assegura unicidade dos registros, desempenho otimizado e acesso rápido às informações, independentemente do volume de dados armazenados.

O DynamoDB, por sua própria arquitetura distribuída e serverless, oferece altíssima disponibilidade, replicação automática dos dados e um desempenho consistente, com respostas que operam em escala de milissegundos. Isso é fundamental para aplicações de controle de estoque, que demandam não apenas precisão nas informações, mas também respostas rápidas e confiáveis em tempo real.

O desenvolvimento do Turtle Track foi planejado de forma modular, separando claramente as responsabilidades entre a camada de apresentação e a camada de dados. Na camada de apresentação, foi utilizada a biblioteca Tkinter, responsável pela construção de uma interface gráfica simples, intuitiva e funcional, contendo elementos como campos de entrada, botões de ação e navegação por abas. As abas foram organizadas para diferentes finalidades operacionais, incluindo tela de acesso, menu administrativo, cadastro de produtos, busca, edição e remoção de itens do estoque.

Em segundo plano, a camada de integração com o DynamoDB é composta por funções específicas e bem definidas. Cada função foi projetada para realizar uma operação determinada sobre a base de dados. Dentre elas, destacam-se as funções de verificação de existência de um produto, que impedem o cadastro de itens duplicados, inserção de novos produtos, consulta de informações por meio do ID, edição de dados existentes, além da exclusão segura de registros, garantindo sempre a integridade e a consistência dos dados.

O backend da aplicação foi otimizado com o uso de expressões condicionais, filtragem inteligente de dados e atualizações direcionadas, recursos nativos do DynamoDB que garantem não apenas eficiência operacional, mas também redução no custo de processamento, já que o DynamoDB adota um modelo de cobrança baseado em capacidade provisionada ou sob demanda, conforme o uso.

Essa arquitetura híbrida — interface local com banco de dados na nuvem — proporciona uma experiência de usuário fluida e de alta performance, eliminando limitações comuns em soluções que dependem de arquivos locais ou bancos de dados relacionais que exigem infraestrutura robusta e configuração complexa.

Diante disso, a seguir será apresentado o código-fonte completo da aplicação Turtle Track, contendo todas as estruturas, funções e lógicas de interface e integração com o DynamoDB. Este

código materializa, de forma prática, todos os conceitos, arquiteturas e modelagens discutidos ao longo deste trabalho, permitindo uma visão detalhada do funcionamento interno do sistema e sua aplicabilidade no contexto real de controle de estoque.

```
from boto3 import resource

from boto3.dynamodb.conditions import Attr, Key

from datetime import datetime

import tkinter as tk

from tkinter import ttk, messagebox

from PIL import Image, ImageTk

from decimal import Decimal



dynamodb = resource('dynamodb', region_name="sa-east-1")   # Substitua
pela sua região

ProdTable = dynamodb.Table('TurtleTrackDB')

UserTable = dynamodb.Table('Users')



def verificar_id_existente(product_id):

    response = ProdTable.get_item(Key={'product_id': product_id})

    return 'Item' in response



def verificar_nome_existente(nome):

    response = ProdTable.scan(

        FilterExpression=Attr('nome').eq(nome)

    )

    return len(response.get('Items', [])) > 0



def salvar_produto_bd(product_id, nome, quantidade, preco):
```

```
    response = ProdTable.put_item(
        Item={
            'product_id': product_id,
            'nome': nome,
            'quantidade': quantidade,
            'preco': preco,
            'data_criacao': datetime.now().isoformat()
        }
    )
    print("Produto salvo:", response)
    return response


def buscar_produto_bd(product_id):
    response = ProdTable.get_item(
        Key={
            'product_id': product_id
        }
    )
    print("Produto:", response)
    return response.get('Item')


def edit_produto_bd(product_id, nome, quantidade, preco):
    response = ProdTable.update_item(
        Key={'product_id': product_id},
        UpdateExpression="set nome=:n, quantidade=:q, preco=:p",
```

```
        ExpressionAttributeValues={
            ':n': nome,
            ':q': quantidade,
            ':p': preco
        },
        ReturnValues="UPDATED_NEW"
    )
    print("Produto atualizado:", response)
    return response

def delete_produto_bd(product_id):
    response = ProdTable.delete_item(
        Key={'product_id': product_id}
    )
    print("Produto deletado:", response)

#FUNÇÕES DO CÓDIGO - INTEGRANDO TELA/BANCO

def configurar_abas():
    nivel_acesso = cod_acess_entry.get()

    # Exibe apenas as abas permitidas
    if nivel_acesso == '01':  # Usuário com acesso limitado
        notebook.add(aba_buscar, text="Menu Usuário")
        notebook.forget(aba_acesso)
```

```
    elif nivel_acesso == '02':  # Administrador com acesso total
        notebook.add(aba_menu, text="Menu Admin")
        notebook.forget(aba_acesso)

#Função botão add da busca
def chama_add():
    notebook.add(aba_adicionar, text="Adicionar Produto")
    notebook.forget(aba_buscar)

#Função botão busca do menu-admin
def chama_buscar_admin():
    notebook.add(aba_buscar, text="Buscar Produto")
    notebook.forget(aba_menu)

#Função botão edit do menu-admin
def chama_edit_admin():
    notebook.add(aba_edit, text="Editar Produto")
    notebook.forget(aba_menu)

#Função botão fechar da aba add
def exit_add():
    notebook.forget(aba_adicionar)
    notebook.add(aba_acesso, text="Acesso")

#Função botão fechar da aba busca
```

```
def exit_busca():
    notebook.forget(aba_buscar)
    notebook.add(aba_acesso, text="Acesso")

#Função botão fechar da menu
def exit_menu():
    notebook.forget(aba_menu)
    notebook.add(aba_acesso, text="Acesso")

#Função botão fechar da aba edit
def exit_edit():
    notebook.forget(aba_edit)
    notebook.add(aba_acesso, text="Acesso")

# Função para obter dados da interface e adicionar produto ao estoque
def adicionar_produto_interface():
    produto_id = int(cod_entry.get())
    nome = name_entry.get()
    try:
        quantidade = Decimal(qtd_entry.get())
        preco = Decimal(preco_entry.get())
    except ValueError:
            messagebox.showerror("Erro", "Quantidade e preço devem ser números válidos.")
        return
```

```
    # Verificar se o ID já existe
    if verificar_id_existente(produto_id):
          messagebox.showerror("Erro", "ID já cadastrado. Utilize outro
ID.")
        return

    # Verificar se o nome já existe
    if verificar_nome_existente(nome):
         messagebox.showerror("Erro", "Nome já cadastrado. Utilize outro
nome.")
        return

    try:
        # Jogando dados pra função add BD
            response = salvar_produto_bd(produto_id, nome, quantidade,
preco)

        if response['ResponseMetadata']['HTTPStatusCode'] == 200:

                    messagebox.showinfo("Sucesso", "Produto salvo com
sucesso.")
            # Limpa os campos da interface
            cod_entry.delete(0, tk.END)
            name_entry.delete(0, tk.END)
            qtd_entry.delete(0, tk.END)
            preco_entry.delete(0, tk.END)
```

```
        else:
            messagebox.showerror("Erro", "Falha ao salvar produto.")
    except:
        messagebox.showinfo("Erro", "Falha ao salvar produto.")


# Função para buscar e exibir produto da aba busca
def busca_produto_interface_busca():
    produto_id = int(cod_busca_entry.get())
    produto = buscar_produto_bd(produto_id)

    if produto:
        nome_busca_entry.config(state='normal')  # Permite edição
        nome_busca_entry.delete(0, tk.END)
         nome_busca_entry.insert(0, produto.get('nome', ''))  # Nome do
produto
        nome_busca_entry.config(state='readonly')  # Só leitura

        qtd_busca_entry.config(state='normal')
        qtd_busca_entry.delete(0, tk.END)
          qtd_busca_entry.insert(0, str(produto.get('quantidade', '')))
# Quantidade
        qtd_busca_entry.config(state='readonly')

        preco_busca_entry.config(state='normal')
        preco_busca_entry.delete(0, tk.END)
          preco_busca_entry.insert(0, str(produto.get('preco', '')))  #
```

```
Preço

        preco_busca_entry.config(state='readonly')

    else:

         messagebox.showerror("Erro", f"Produto com código {produto_id}
não encontrado.")



#Função para buscar e exibir produto da aba edit

def busca_produto_interface_edit():

    produto_id = int(cod_edit_entry.get())

    produto = buscar_produto_bd(produto_id)



    if produto:

        # Exibir os dados do produto na interface de busca

        nome_edit_entry.config(state='normal')  # Permite edição

        nome_edit_entry.delete(0, tk.END)

        nome_edit_entry.insert(0, produto.get('nome', ''))  # Nome

             nome_edit_entry.config(state='readonly')    # Modo somente
leitura



        qtd_edit_entry.config(state='normal')

        qtd_edit_entry.delete(0, tk.END)

         qtd_edit_entry.insert(0, str(produto.get('quantidade', '')))  #
Quantidade

        qtd_edit_entry.config(state='readonly')



        preco_edit_entry.config(state='normal')
```

```
        preco_edit_entry.delete(0, tk.END)

           preco_edit_entry.insert(0, str(produto.get('preco', '')))   # Preço

        preco_edit_entry.config(state='readonly')

    else:

         messagebox.showerror("Erro", f"Produto com código {produto_id} não encontrado.")



    nome_edit_entry.config(state='normal')  # Permite edição

    qtd_edit_entry.config(state='normal')

    preco_edit_entry.config(state='normal')




# Função para buscar e atualizar o produto

def editar_produto_interface():

    produto_id = int(cod_edit_entry.get())

    nome = nome_edit_entry.get()

    try:

        quantidade = Decimal(qtd_edit_entry.get())

        preco = Decimal(preco_edit_entry.get())

    except ValueError:

            messagebox.showerror("Erro", "Quantidade e preço devem ser números válidos.")

        return



    # Verifica se o ID existe antes de editar
```

```
    if not verificar_id_existente(produto_id):
            messagebox.showerror("Erro", "Produto não encontrado para editar.")
        return

    # Verifica se o nome já está em outro produto
    produtos_com_nome = ProdTable.scan(
        FilterExpression=Attr('nome').eq(nome)
    ).get('Items', [])

    if produtos_com_nome:
        for produto in produtos_com_nome:
            if produto['product_id'] != produto_id:
                messagebox.showerror("Erro", "Nome já cadastrado em outro produto.")
                return

    # Jogando dados pra função edit BD
    response = edit_produto_bd(produto_id, nome, quantidade, preco)

    if response and 'Attributes' in response:
            messagebox.showinfo("Sucesso", "Produto atualizado com sucesso.")

        # Limpa os campos da interface de edição
        cod_edit_entry.delete(0, tk.END)
```

```
        nome_edit_entry.delete(0, tk.END)
        qtd_edit_entry.delete(0, tk.END)
        preco_edit_entry.delete(0, tk.END)
    else:
        messagebox.showerror("Erro", "Falha ao atualizar o produto.")



# Função para buscar e excluir o produto
def deletar_produto_interface():
    produto_id = int(cod_edit_entry.get())
    # Jogando dados pra função delete BD
    response = delete_produto_bd(produto_id)


    if response and 'Attributes' in response:
        messagebox.showinfo("Erro", "Falha ao deletar produto.")
    else:
        messagebox.showinfo("Sucesso", "Produto deletado com sucesso.")
        # Limpa os campos da interface
        cod_edit_entry.delete(0, tk.END)
        nome_edit_entry.config(state='normal')
        nome_edit_entry.delete(0, tk.END)
        qtd_edit_entry.config(state='normal')
        qtd_edit_entry.delete(0, tk.END)
        preco_edit_entry.config(state='normal')
        preco_edit_entry.delete(0, tk.END)
```

```python
#INTERFACE - TELAS


# Cores e estilos
COR_FUNDO = "#f0f4f7"
COR_BOTAO = "#4CAF50"
COR_TEXTO = "#333"
COR_DESTAQUE = "#2E8B57"
FONTE = ("Arial", 10)
FONTE_TITULO = ("Arial", 14, "bold")


# Início da interface
tela = tk.Tk()
tela.title("Turtle Track - Controle de Estoque")
tela.configure(bg=COR_FUNDO)


# Logo
logo_img = Image.open("98efd7f0-09a9-4033-a446-7a637d6903e2.png")
logo_img = logo_img.resize((200, 80))
logo = ImageTk.PhotoImage(logo_img)
logo_label = tk.Label(tela, image=logo, bg=COR_FUNDO)
logo_label.pack(pady=5)


# Notebook
```

```
notebook = ttk.Notebook(tela)

notebook.pack(pady=10, expand=True)



style = ttk.Style()

style.configure("TNotebook", background=COR_FUNDO)

style.configure("TFrame", background=COR_FUNDO)

style.configure("TLabel", background=COR_FUNDO, font=FONTE)

style.configure("TButton", font=FONTE)



# Aba de Acesso

aba_acesso = ttk.Frame(notebook)

notebook.add(aba_acesso, text="Acesso")



title_label = tk.Label(aba_acesso, text="Acesso ao Sistema",
font=FONTE_TITULO, bg=COR_FUNDO, fg=COR_DESTAQUE)

title_label.grid(row=0, column=1, padx=10, pady=10)



cod_acess_label = tk.Label(aba_acesso, text="Código do Acesso:",
bg=COR_FUNDO)

cod_acess_label.grid(row=1, column=0, padx=10, pady=6)



cod_acess_entry = tk.Entry(aba_acesso)

cod_acess_entry.grid(row=2, column=1, padx=10, pady=6)



botao_acess = tk.Button(aba_acesso, text="🔓 Acessar", bg=COR_BOTAO,
fg="white", command=configurar_abas)
```

```
botao_acess.grid(row=3, column=2, pady=10)



# Aba Menu Admin

aba_menu = ttk.Frame(notebook)

notebook.add(aba_menu, text="Menu - Admin")



botao_busca_menu_admin = tk.Button(aba_menu, text="🔍 Buscar Produto",
bg=COR_BOTAO, fg="white", command=chama_buscar_admin)

botao_busca_menu_admin.grid(row=0, column=0, columnspan=2, pady=10)



botao_edit_menu_admin = tk.Button(aba_menu, text="✏️ Editar Produto",
bg=COR_BOTAO, fg="white", command=chama_edit_admin)

botao_edit_menu_admin.grid(row=0, column=4, columnspan=2, pady=10)



botao_fechar_busca = tk.Button(aba_menu, text="❌ Fechar", bg="red",
fg="white", command=exit_menu)

botao_fechar_busca.grid(row=1, column=2, columnspan=2, pady=10)



# Aba Buscar Produto

aba_buscar = ttk.Frame(notebook)

notebook.add(aba_buscar, text="Buscar Produto")



botao_fechar_busca = tk.Button(aba_buscar, text="❌", bg="red",
fg="white", command=exit_busca)

botao_fechar_busca.grid(row=0, column=3, columnspan=2, pady=10)



tk.Label(aba_buscar, text="Código do Produto:",
```

```
bg=COR_FUNDO).grid(row=0, column=0, padx=10, pady=6)

cod_busca_entry = tk.Entry(aba_buscar)

cod_busca_entry.grid(row=0, column=1, padx=10, pady=6)



tk.Button(aba_buscar,    text="🔍    Buscar    Produto",    bg=COR_BOTAO,
fg="white",           command=busca_produto_interface_busca).grid(row=1,
column=1, columnspan=2, pady=10)



tk.Label(aba_buscar, text="Nome do Produto:", bg=COR_FUNDO).grid(row=2,
column=0, padx=10, pady=6)

nome_busca_entry = tk.Entry(aba_buscar, state='readonly')

nome_busca_entry.grid(row=2, column=1, padx=10, pady=6)



tk.Label(aba_buscar,        text="Quantidade        do        Produto:",
bg=COR_FUNDO).grid(row=3, column=0, padx=10, pady=6)

qtd_busca_entry = tk.Entry(aba_buscar, state='readonly')

qtd_busca_entry.grid(row=3, column=1, padx=10, pady=6)



tk.Label(aba_buscar,          text="Preço          do          Produto:",
bg=COR_FUNDO).grid(row=4, column=0, padx=10, pady=6)

preco_busca_entry = tk.Entry(aba_buscar, state='readonly')

preco_busca_entry.grid(row=4, column=1, padx=10, pady=6)



tk.Button(aba_buscar,   text="➕   Adicionar   Produto",   bg="#2196F3",
fg="white",   command=chama_add).grid(row=5,   column=1,   columnspan=2,
pady=10)



# Aba Adicionar Produto
```

```
aba_adicionar = ttk.Frame(notebook)

notebook.add(aba_adicionar, text="Adicionar Produto")



tk.Button(aba_adicionar,      text="❌",      bg="red",      fg="white",
command=exit_add).grid(row=0, column=3, columnspan=2, pady=10)

tk.Label(aba_adicionar,        text="Código        do        Produto:",
bg=COR_FUNDO).grid(row=0, column=0, padx=10, pady=6)

cod_entry = tk.Entry(aba_adicionar)

cod_entry.grid(row=0, column=1, padx=10, pady=6)



tk.Label(aba_adicionar,        text="Nome        do        Produto:",
bg=COR_FUNDO).grid(row=1, column=0, padx=10, pady=6)

name_entry = tk.Entry(aba_adicionar)

name_entry.grid(row=1, column=1, padx=10, pady=6)



tk.Label(aba_adicionar,       text="Quantidade       do       Produto:",
bg=COR_FUNDO).grid(row=2, column=0, padx=10, pady=6)

qtd_entry = tk.Entry(aba_adicionar)

qtd_entry.grid(row=2, column=1, padx=10, pady=6)



tk.Label(aba_adicionar,        text="Preço        do        Produto:",
bg=COR_FUNDO).grid(row=3, column=0, padx=10, pady=6)

preco_entry = tk.Entry(aba_adicionar)

preco_entry.grid(row=3, column=1, padx=10, pady=6)



tk.Button(aba_adicionar,  text="➕  Adicionar  Produto",  bg=COR_BOTAO,
fg="white",  command=adicionar_produto_interface).grid(row=4,  column=0,
columnspan=2, pady=10)
```

```
# Aba Editar Produto

aba_edit = ttk.Frame(notebook)

notebook.add(aba_edit, text="Editar Produto")



tk.Button(aba_edit,        text="❌",        bg="red",        fg="white",
command=exit_edit).grid(row=0, column=3, columnspan=2, pady=10)

tk.Label(aba_edit, text="Código do Produto:", bg=COR_FUNDO).grid(row=0,
column=0, padx=10, pady=6)

cod_edit_entry = tk.Entry(aba_edit)

cod_edit_entry.grid(row=0, column=1, padx=10, pady=6)



tk.Button(aba_edit, text="🔍 Buscar Produto", bg=COR_BOTAO, fg="white",
command=busca_produto_interface_edit).grid(row=1,              column=1,
columnspan=2, pady=10)



tk.Label(aba_edit, text="Nome do Produto:", bg=COR_FUNDO).grid(row=2,
column=0, padx=10, pady=6)

nome_edit_entry = tk.Entry(aba_edit)

nome_edit_entry.grid(row=2, column=1, padx=10, pady=6)



tk.Label(aba_edit,         text="Quantidade         do         Produto:",
bg=COR_FUNDO).grid(row=3, column=0, padx=10, pady=6)

qtd_edit_entry = tk.Entry(aba_edit)

qtd_edit_entry.grid(row=3, column=1, padx=10, pady=6)



tk.Label(aba_edit, text="Preço do Produto:", bg=COR_FUNDO).grid(row=4,
column=0, padx=10, pady=6)
```

```
preco_edit_entry = tk.Entry(aba_edit)

preco_edit_entry.grid(row=4, column=1, padx=10, pady=6)



tk.Button(aba_edit, text="✏️ Editar Produto", bg="#FFA000", fg="white",
command=editar_produto_interface).grid(row=5, column=0, pady=10)

tk.Button(aba_edit,    text="🗑️    Excluir    Produto",    bg="darkred",
fg="white",   command=deletar_produto_interface).grid(row=5,   column=3,
pady=10)



# Oculta as abas

notebook.forget(aba_edit)

notebook.forget(aba_adicionar)

notebook.forget(aba_buscar)

notebook.forget(aba_menu)



tela.mainloop()
```

5.6 Análise dos Resultados e Benefícios Observados

A adoção do DynamoDB na arquitetura do Turtle Track proporcionou ganhos significativos em desempenho, confiabilidade e simplicidade de operação. O modelo serverless elimina a necessidade de gerenciamento de servidores, backups manuais ou escalonamento de infraestrutura, permitindo que o foco da operação seja na lógica de negócio e não na manutenção do sistema.

Do ponto de vista do usuário, a interface desenvolvida com Tkinter garante uma curva de aprendizado extremamente baixa, permitindo que até usuários sem conhecimento técnico consigam operar o sistema de maneira eficiente.

A flexibilidade oferecida pelo DynamoDB também se mostra vantajosa na possibilidade de expansão futura do sistema, seja adicionando funcionalidades como controle de fornecedores, geração de relatórios, dashboards analíticos ou integração com APIs externas.

5.7 Considerações Finais da Aplicação

O desenvolvimento do Turtle Track demonstra como a combinação de tecnologias modernas, como Python e DynamoDB, permite criar soluções robustas, escaláveis e de fácil utilização para problemas comuns no mercado, como o controle de estoque. A aplicação atende plenamente às necessidades propostas, proporcionando um gerenciamento eficiente dos produtos, redução de erros

manuais e aumento da produtividade.

O sistema ainda possui grande potencial de expansão, podendo evoluir para aplicações web, integração com serviços móveis, dashboards interativos e automação de processos logísticos. Dessa forma, o Turtle Track representa não apenas uma solução prática, mas também uma base sólida para futuros aprimoramentos no contexto da transformação digital de negócios.

6.Conclusão

O desenvolvimento deste trabalho proporcionou uma compreensão aprofundada sobre os bancos de dados não relacionais, suas características, modelos de dados e, especialmente, sobre o funcionamento do Amazon DynamoDB. A análise teórica realizada evidenciou que os bancos NoSQL surgiram como resposta às limitações dos modelos relacionais, oferecendo maior escalabilidade, flexibilidade e performance em cenários com alta demanda de dados e necessidade de respostas rápidas.

A escolha pelo DynamoDB se mostrou tecnicamente justificável, dado seu modelo serverless, sua capacidade de escalabilidade automática e sua baixa latência, fatores indispensáveis para aplicações modernas que operam em tempo real. A integração do DynamoDB com a linguagem Python, por meio da biblioteca Boto3, demonstrou ser não apenas viável, mas também eficiente, permitindo construir uma solução robusta e de fácil manutenção, alinhada às melhores práticas do desenvolvimento na nuvem.

A aplicação Turtle Track, desenvolvida como parte prática deste projeto, materializou os conceitos estudados, provando que é possível criar soluções eficientes para controle de estoque utilizando tecnologias baseadas em bancos de dados NoSQL. O sistema atendeu plenamente às necessidades propostas, possibilitando cadastro, consulta, atualização e exclusão de produtos de forma rápida, segura e escalável. Além disso, o modelo arquitetônico adotado oferece uma base sólida para expansões futuras, como integração com APIs, desenvolvimento de dashboards e migração para soluções web ou mobile.

Por fim, este trabalho não apenas proporcionou o desenvolvimento de uma aplicação funcional, mas também consolidou o entendimento de como a transformação digital, impulsionada por bancos de dados não relacionais e arquiteturas em nuvem, está moldando o futuro da gestão de dados. A adoção de soluções como o DynamoDB representa não apenas uma escolha tecnológica, mas uma estratégia de negócio voltada à escalabilidade, segurança, eficiência e inovação. Portanto, conclui-se que a utilização de bancos NoSQL, quando bem planejada e aplicada aos cenários corretos, oferece vantagens significativas frente aos modelos tradicionais, representando um caminho natural e cada vez mais necessário para organizações que buscam competitividade na era digital.

Referências Bibliográficas

SADALAGE, Pramod J.; FOWLER, Martin. *NoSQL Essencial: Um Guia Conciso para o Mundo Emergente da Persistência Poliglota*. São Paulo: Novatec Editora, 2013.Livros SBC+4Google Livros+4novatec.com.br+4

CHODOROW, Kristina. *MongoDB: The Definitive Guide*. 3. ed. Sebastopol: O'Reilly Media, 2019.

KLEPPMANN, Martin. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol: O'Reilly Media, 2017.

SOTTO, Eder Carlos Salazar. Comparativo entre os modelos de banco de dados relacional e não relacional. *Interface Tecnológica*, v. 16, n. 2, 2019. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/673. Acesso em: 20 maio 2025.ResearchGate+1Revista Fatec TQ+1

PANIZ, David. *NoSQL: Banco de Dados para Aplicações Modernas*. São Paulo: Casa do Código, 2014.casadocodigo.com.br

FEITOSA, Márcio Porto. *MongoDB - o banco de dados NoSQL mais utilizado da atualidade: Uma introdução com instalação, comandos e aplicações em Java*. São Paulo: Clube de Autores, 2020.Amazon Brasil+1Clube de Autores+1

AZURE Architecture Center. Dados não relacionais. Microsoft Learn. Disponível em: https://learn.microsoft.com/pt-br/azure/architecture/data-guide/big-data/non-relational-data. Acesso em: 03 maio 2025.

ScyllaDB. Introduction to DynamoDB. Disponível em: https://www.scylladb.com/learn/dynamodb/introduction-to-dynamodb/. Acesso em: 14 maio 2025.

Amazon Web Services. Snap Inc. melhora latência e reduz custos utilizando Amazon DynamoDB. Disponível em: https://aws.amazon.com/pt/solutions/case-studies/snap-dynamodb/?pg=dynamodb&sec=cs#SnapChat. Acesso em: 14 maio 2025.

SprinkleData. MySQL vs DynamoDB: A Comprehensive Comparison. Disponível em: https://www.sprinkledata.com/blogs/mysql-vs-dynamodb-a-comprehensive-comparison. Acesso em: 14 maio 2025.

PANOPLY. DynamoDB vs MongoDB: Performance Comparison & Cost Analysis. Disponível em: https://blog.panoply.io/dynamodb-vs-mongodb. Acesso em: 20 maio 2025.blog.panoply.io

MongoDB Inc. Comparing DynamoDB And MongoDB. Disponível em: https://www.mongodb.com/resources/compare/mongodb-dynamodb. Acesso em: 20 maio

2025.MongoDB

BYTEBASE. DynamoDB vs. MongoDB: a Complete Comparison in 2025. Disponível em: https://www.bytebase.com/blog/dynamodb-vs-mongodb/. Acesso em: 20 maio 2025.BytebaseProjectPro

BMC. DynamoDB vs MongoDB: Comparing NoSQL Databases. Disponível em: https://www.bmc.com/blogs/mongodb-vs-dynamodb/. Acesso em: 20 maio 2025.BMC

KINSTA. DynamoDB vs MongoDB: Choose One and Say No to SQL. Disponível em: https://kinsta.com/blog/dynamodb-vs-mongodb/. Acesso em: 20 maio 2025.Kinsta®

PROJECTPRO. DynamoDB Vs. MongoDB: Battle Of The Best Databases. Disponível em: https://www.projectpro.io/article/dynamodb-vs-mongodb/826. Acesso em: 20 maio 2025.CloudZero+1ProjectPro+1

CLOUDZERO. DynamoDB Vs. MongoDB: Battle Of The Best Databases. Disponível em: https://www.cloudzero.com/blog/dynamodb-vs-mongodb/. Acesso em: 20 maio 2025.CloudZero

INSTACLUSTR. 10 amazing use cases for NoSQL in 2025. Disponível em: https://www.instaclustr.com/education/nosql-database/10-amazing-use-cases-for-nosql-in-2025/. Acesso em: 20 maio 2025.Instaclustr

BAREMON. Database trends of 2025: Rankings, new technologies and changes. Disponível em: https://www.baremon.eu/database-trends-of-2025/. Acesso em: 20 maio 2025.Baremon | Data Experts

BUSINESS RESEARCH INSIGHTS. Tamanho do mercado NoSQL, compartilhamento, crescimento e tendências. Disponível em: https://www.businessresearchinsights.com/pt/market-reports/nosql-market-111657. Acesso em: 20 maio 2025.

Padhy, R. P., Patra, M. R., & Satapathy, S. C. RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Databases. International Journal of Advanced Engineering Sciences and Technologies (IJAEST), vol. 11, no. 1, pp. 15–30, 2011. Disponível em: https://arxiv.org/abs/1201.6597. Acesso em: 20 maio 2025.

DeCandia, G. et al. Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP), 2007. Disponível em: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf. Acesso em: 20 maio 2025.

Microsoft. Dados não relacionais – Azure Architecture Center. Disponível em: https://learn.microsoft.com/pt-br/azure/architecture/data-guide/big-data/non-relational-data. Acesso em: 03 maio 2025, às 16:03.