GOVERNO DO ESTADO DA PARAÍBA
SECRETARIA DE ESTADO DA CIÊNCIA,
TECNOLOGIA, INOVAÇÃO E ENSINO SUPERIOR -
SECTIES UNIVERSIDADE ESTADUAL DA PARAÍBA
CENTRO DE CIÊNCIAS BIOLÓGICAS E SOCIAIS
APLICADAS – CCBSA CAMPUS V JOÃO PESSOA – PB CURSO SUPERIOR EM
CIÊNCIA DE DADOS

DJEFFERSON DOS SANTOS LIMA
LUCAS EDSON DA SILVA COSENDEY
LUAN TORRES ARRUDA
NATHALIA RAYSSA DE OLIVEIRA CARDOSO
RIANDERSON LIMA DOS SANTOS

APACHE CASSANDRA

Justificativa Técnica para Sistema de Mensagens

João Pessoa
2025

1. INTRODUÇÃO

O presente documento tem por objetivo estabelecer a fundamentação teórica e técnica que alicerça a escolha do Apache Cassandra como o Sistema Gerenciador de Banco de Dados (SGBD) principal para a solução de mensageria em tempo real proposta. A decisão arquitetural deriva de uma análise rigorosa dos requisitos não funcionais do sistema, especificamente no que tange à escalabilidade horizontal, alta disponibilidade e latência de escrita. O cenário de mensageria instantânea impõe desafios de volumetria e concorrência que as arquiteturas de bancos de dados relacionais (RDBMS) tradicionais, baseadas no modelo ACID estrito, tendem a não atender eficientemente em alta escala.

2. FUNDAMENTAÇÃO TEÓRICA: BANCOS DE DADOS COLUNARES

2.1 . Apache Cassandra: A Abordagem Wide-Column Store

A principal diferença entre um banco de dados tradicional (SQL) e um banco de dados colunar reside na forma como os dados são fisicamente armazenados no disco.

- Bancos Tradicionais (Orientados a Linha): Armazenam os dados sequencialmente por registro (linha). Todos os dados de uma linha (ID, nome, email, etc.) são escritos juntos em um bloco. Este modelo é otimizado para cargas de trabalho transacionais (OLTP), onde é comum ler ou atualizar um registro inteiro de uma vez (ex: "buscar todos os dados do usuário com ID 5").
- Bancos Colunares (Orientados a Coluna): Armazenam os dados por coluna. Todos os valores de uma mesma coluna (ex: todos os "nomes") são escritos juntos sequencialmente.

Esta abordagem [colunar] é extremamente eficiente para consultas analíticas (OLAP), pois uma consulta que agrega dados de poucas colunas (ex: SUM(valor_venda)) precisa ler apenas esses blocos de dados, ignorando todas as outras colunas. Isso resulta em uma drástica redução de I/O (Entrada/Saída) de disco e permite altíssimos níveis de compressão, já que dados do mesmo tipo são armazenados juntos (HEWITT, 2010).

É crucial entender que o Apache Cassandra, embora frequentemente chamado de colunar, não é um banco de dados colunar puro (como o Google BigQuery ou Amazon Redshift). Tecnicamente, o Cassandra é um "wide-column store" (ou armazém de colunas largas).

Ele combina a eficiência do armazenamento colunar com um modelo de dados de chave-valor. Em vez de uma tabela rígida, ele usa "famílias de colunas" (Column Families), que funcionam como um mapa de mapas (FOWLER; SADALAGE, 2012). Cada linha é identificada por uma chave única (Row Key) e pode ter um número virtualmente ilimitado e dinâmico de colunas, agrupadas e ordenadas por "chaves de clusterização" (Clustering Keys). Isso se encaixa perfeitamente no nosso caso de uso:

- Na tabela mensagens, a conversa_id atua como a Row Key (Chave de Partição), agrupando fisicamente todas as mensagens de uma conversa no mesmo nó (ou nós de réplica).
- O timestamp (nosso TIMEUUID) atua como a Clustering Key, ordenando fisicamente as mensagens no disco dentro de cada partição.

Quando buscamos as mensagens de uma conversa, o Cassandra vai diretamente à partição correta e lê as mensagens em ordem cronológica, uma operação de leitura sequencial de disco extremamente rápida, sem a necessidade de índices B-tree custosos como no SQL.

3. APACHE CASSANDRA: HISTÓRIA E DEFINIÇÃO

3.1 Origem

O Apache Cassandra foi originalmente desenvolvido no Facebook por Avinash Lakshman (um dos autores do Amazon Dynamo) e Prashant Malik, com o objetivo de potencializar o recurso de busca da caixa de entrada (Inbox Search). O desafio era gerenciar um volume massivo de mensagens (escritas) com alta disponibilidade e distribuição geográfica (LAKSHMAN; MALIK, 2010).

Foi lançado como código aberto em 2008 e sua arquitetura é uma síntese brilhante de dois artigos seminais:

1. Google Bigtable: De onde herda seu modelo de dados (o wide-column store, famílias de colunas, e o uso de SSTables - Sorted String Tables - para armazenamento imutável em disco).

2. Amazon Dynamo: De onde herda sua arquitetura de distribuição (modelo peer-to-peer descentralizado, particionamento consistente via hash ring, replicação e consistência eventual).

O Cassandra é, portanto, um banco de dados NoSQL, distribuído, descentralizado (sem mestre), tolerante a falhas e de alta disponibilidade.

Sua arquitetura "masterless" (sem mestre) é um pilar central. Todos os nós no cluster são idênticos e se comunicam através de um protocolo "Gossip". Não há um "nó mestre" que represente um ponto único de falha. Isso significa que, se um nó falhar, o sistema continua operando normalmente, um requisito não negociável para um sistema de mensagens que precisa estar sempre online.

4. POR QUE CASSANDRA PARA SISTEMAS DE MENSAGENS?

Sistemas de mensagens em tempo real (WhatsApp, Telegram, Instagram) enfrentam desafios específicos:

- Milhões de mensagens por segundo
- Alta disponibilidade (99.999%)
- Latência baixa mesmo com bilhões de registros
- Crescimento imprevisível
- Distribuição geográfica global

Para o nosso projeto, a escolha do Cassandra sobre um SQL tradicional se justifica pela necessidade de suportar milhões de mensagens sendo enviadas simultaneamente, distribuídas globalmente, sem que o banco de dados se torne um gargalo de escrita ou de escalabilidade.

5. LIMITAÇÕES DE BANCOS SQL TRADICIONAIS

5.1 Performance em Escritas Massivas

Problema: Bancos relacionais utilizam locks (bloqueios) em operações INSERT. Com milhões de mensagens simultâneas, isso cria contenção severa.

Impacto: Sistema lento ou travamento em horários de pico.
Cassandra: Escritas append-only (LSM-tree), sem locks. Retorno quase instantâneo.

5.2 Escalabilidade Horizontal

Problema: Escalar SQL horizontalmente requer sharding manual, complexo e custoso.
Impacto: Reengenharia constante, downtimes para manutenção.
Cassandra: Adicionar nós = escalabilidade linear automática, sem downtime.

5.3 Ordenação Temporal

Problema: SQL requer índices B-tree custosos para ordenar por timestamp.
Impacto: Performance degrada com volume de dados.
Cassandra: Clustering Key ordena dados fisicamente no disco (nativo).

5.4 Single Point of Failure

Problema: Arquiteturas Master-Slave têm ponto único de falha.
Impacto: Se o master cai, o sistema para completamente.
Cassandra: Arquitetura peer-to-peer, sem master. Falha de nó não afeta operação.

6. COMPARAÇÃO PRÁTICA: SQL VS CASSANDRA

Cenário	SQL Tradicional	Apache Cassandra
Inserir 1M msgs/seg	Locks, lentidão, travamento	Linear, sem degradação
Escalar 1TB → 10TB	Sharding manual, downtime	Adiciona nós, zero downtime
Últimas 50 msgs	Index scan custoso	Leitura de partição O(1)
Falha de servidor	Sistema para	Sistema continua operando
Multi-datacenter	Replicação manual complexa	Nativo, automático

7. CASOS DE USO REAIS EM PRODUÇÃO

WhatsApp

- ~100 bilhões de mensagens/dia
- Cassandra para histórico de mensagens
- Motivo: Única solução com escala horizontal + latência baixa

Instagram

- Bilhões de posts e comentários
- Cassandra para feed de atividades
- Motivo: Ordenação temporal nativa

Netflix

- 230 milhões de usuários
- Cassandra para histórico de visualizações
- Motivo: Leituras rápidas mesmo com petabytes

8. PONTOS POSITIVOS (VANTAGENS)

Desempenho de Escrita Extremo: O Cassandra utiliza uma arquitetura de Log-Structured Merge-Tree (LSM-tree). Uma escrita apenas anexa dados a um log (Commit Log) e a uma tabela em memória (Memtable), retornando sucesso ao cliente quase instantaneamente. As escritas nunca modificam dados no lugar, evitando o overhead de busca e bloqueio (locks) dos RDBMS (HEWITT, 2010).

Escalabilidade Horizontal Linear: Para suportar mais usuários ou mais mensagens, basta adicionar mais nós (servidores comuns) ao cluster. O Cassandra redistribui os dados automaticamente (rebalanceamento). O desempenho escala linearmente com o número de nós, sem necessidade de downtime.

Alta Disponibilidade e Tolerância a Falhas: Sua arquitetura sem mestre (masterless) e a replicação de dados (configurável via Replication Factor) entre múltiplos nós garantem que o sistema permaneça operacional mesmo durante falhas de hardware ou de rede.

Suporte Multi-Datacenter Nativo: O Cassandra foi projetado desde o início para operar em múltiplos data centers, permitindo distribuição geográfica de dados, latência mais baixa para usuários globais e recuperação de desastres robusta.

Flexibilidade de Esquema: Sendo NoSQL, podemos adicionar novas "colunas" (como status de "lida", "editada", etc.) às mensagens dinamicamente, sem alterar o esquema da tabela inteira ou realizar migrações custosas.

9. PONTOS NEGATIVOS (DESVANTAGENS)

Apesar de suas vantagens para o nosso caso de uso, o Cassandra possui limitações que devem ser gerenciadas:

Consistência Eventual (Eventual Consistency): Por padrão, o Cassandra prioriza a Disponibilidade e a Tolerância a Particionamento (AP, segundo o Teorema CAP) em detrimento da Consistência forte. Mitigação: O Cassandra oferece "Consistência Sintonizável" (Tunable Consistency). Podemos exigir que uma leitura ou escrita seja confirmada por um QUORUM (maioria) dos nós, alcançando forte consistência ao custo de maior latência.

Modelagem de Dados Orientada à Consulta: Esta é a maior mudança de paradigma em relação ao SQL. Não se pode fazer JOINs ou consultas GROUP BY complexas. Você deve modelar suas tabelas especificamente para as consultas que sua aplicação fará. A desnormalização de dados não é apenas aceita, é incentivada.

Complexidade Operacional: Gerenciar um cluster Cassandra on-premise é complexo. Requer expertise em tuning da JVM (Java Virtual Machine), estratégias de compactação e processos de reparo de nós. (Nota: Em nosso projeto, mitigamos essa desvantagem utilizando o DataStax Astra DB, que é o Cassandra como um serviço gerenciado).

Desempenho de Leitura (Variável): Embora as leituras pela chave de partição e clusterização sejam extremamente rápidas, as leituras que exigem varredura de muitas partições ou que usam índices secundários são muito ineficientes e devem ser evitadas.

10. CONCLUSÃO

A análise técnica detalhada neste documento corrobora a decisão de adotar o Apache Cassandra como o pilar de persistência para a plataforma de mensageria. A escolha transcende a preferência tecnológica, configurando-se como uma resposta arquitetural mandatória aos limites físicos impostos pelos SGBDs relacionais em cenários de hiperescala.

A superioridade da solução proposta fundamenta-se na transição deliberada do modelo ACID estrito para a consistência eventual (paradigma BASE), alinhando o sistema às restrições do Teorema CAP, onde a Disponibilidade e a Tolerância ao Particionamento são priorizadas para garantir o SLA de 99,999%. A utilização de estruturas de Log-Structured Merge-Trees (LSM) elimina a contenção de locks em escritas concorrentes, enquanto a modelagem baseada em chaves de partição e clusterização assegura recuperações de dados com complexidade algorítmica constante ($O(1)$).

Portanto, a arquitetura distribuída e masterless do Cassandra não apenas mitiga os riscos de gargalos de I/O e pontos únicos de falha, mas provê a elasticidade linear necessária para suportar o crescimento imprevisível da base de usuários, validando a viabilidade técnica e a longevidade do projeto.

11. REFERÊNCIAS

FOWLER, Martin; SADALAGE, Pramod J. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Upper Saddle River, NJ: Addison-Wesley, 2012.

HEWITT, Eben. Cassandra: The Definitive Guide. Sebastopol, CA: O'Reilly Media, 2010.

LAKSHMAN, Avinash; MALIK, Prashant. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, v. 44, n. 2, p. 35-40, 2010.

DATASTAX. Apache Cassandra Documentation. Disponível em: https://cassandra.apache.org/doc/latest/. Acesso em: 18 nov. 2025.