UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

JOSÉ HELTON ALVES PIMENTEL ARAÚJO

MODELAGEM LÓGICA DE BANCO DE DADOS *NOSQL* ORIENTADOS A FAMÍLIA DE COLUNAS COM AML: Uma aplicação prática no Apache Cassandra

Recife
2025

JOSÉ HELTON ALVES PIMENTEL ARAÚJO

MODELAGEM LÓGICA DE BANCO DE DADOS *NOSQL* ORIENTADOS A FAMÍLIA DE COLUNAS COM AML: Uma aplicação prática no Apache Cassandra

Trabalho de Conclusão de Curso apresentado ao Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para a conclusão do curso de Sistemas de Informação, orientado pelo professor Robson do Nascimento Fidalgo.

Orientador (a): Robson Nascimento Fidalgo

Coorientador (a): Gênesis Jeferson Ferreira Pereira de Lima

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Araújo, José Helton Alves Pimentel.
Modelagem lógica de banco de dados NoSQL orientados a família de colunas com AML: Uma aplicação prática no Apache Cassandra / José Helton Alves Pimentel Araújo. - Recife, 2025.
128 p. : il., tab.

Orientador(a): Robson do Nascimento Fidalgo
Cooorientador(a): Gênesis Jeferson Ferreira Pereira de Lima
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Sistemas de Informação - Bacharelado, 2025.
Inclui referências.

1. AML. 2. Modelagem lógica. 3. NoSQL. 4. AML. 5. Agregados. 6. Apache Cassandra. I. Fidalgo, Robson do Nascimento. (Orientação). II. Lima, Gênesis Jeferson Ferreira Pereira de. (Coorientação). IV. Título.

000 CDD (22.ed.)

JOSÉ HELTON ALVES PIMENTEL ARAÚJO

MODELAGEM LÓGICA DE BANCO DE DADOS *NOSQL* ORIENTADOS A FAMÍLIA DE COLUNAS COM AML: Uma aplicação prática no Apache Cassandra

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em 26/03/2025.

Aprovado em: 03 / 04 / 2025

BANCA EXAMINADORA

_____

Prof. Dr. Robson do Nascimento Fidalgo (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Vinícius Cardoso Garcia (Examinador Interno)

Universidade Federal de Pernambuco

RESUMO

Contexto**:* Os bancos de dados (BD) *NoSQL* oferecem soluções para os desafios de *performance* e escalabilidade enfrentados pelos BDs relacionais tradicionais centralizados; no entanto, a sua modelagem apresenta desafios que dificultam sua implementação. Entre os BDs *NoSQL*, destaca-se o orientado à família de colunas, com maior popularidade o Apache Cassandra, devido a sua *performance* com latência de milissegundos em operações de escrita e leitura sobre *Big Data*. ***Problema**:* À medida que mais organizações adotam estes tipos de BDs, surge uma demanda crescente por diretrizes de modelagem lógica específicas. ***Objetivo: Por conseguinte, o objetivo deste trabalho é identificar limitações e avaliar a viabilidade da modelagem lógica no Apache Cassandra, um BDs orientados a família de colunas, utilizando o *Aggregate Modeling Language* (AML), cujos conceitos são fundamentados nos agregados do *Domain-Driven Design* (DDD). ***Método***: Por meio da aplicação de oito abordagens de modelagem lógica utilizando AML a partir do mesmo modelo conceitual no Apache Cassandra. ***Resultados**:* Os testes demonstram que a linguagem AML é uma ferramenta eficiente de modelagem lógica no Apache Cassandra. ***Conclusão**:* Conclui-se que a AML é uma linguagem promissora para modelagem lógica em BDs *NoSQL* orientados à família de colunas, sendo eficaz na definição de agregados e construtores do DDD.

Palavras-chave: *NoSQL*; Modelagem lógica; DDD; AML; Agregados; Família de colunas; Apache Cassandra.

ABSTRACT

Context: NoSQL databases offer solutions to the performance and scalability challenges faced by traditional centralized relational databases; however, their modeling introduces difficulties that hinder implementation. Among NoSQL databases, column-family-oriented models stand out, with Apache Cassandra being the most prominent due to its high performance and millisecond-level latency for read and write operations over Big Data. ***Problem:*** As more organizations adopt these types of databases, there is a growing demand for specific guidelines for logical data modeling. ***Objective:*** Therefore, this work aims to identify limitations and assess the feasibility of logical modeling in Apache Cassandra, a column-family-oriented NoSQL database, using the Aggregate Modeling Language (AML), whose concepts are based on the aggregates of Domain-Driven Design (DDD). ***Method:*** Eight logical modeling approaches using AML were applied based on the same conceptual model in Apache Cassandra. ***Results:*** The experiments demonstrate that AML is an efficient tool for logical modeling in Apache Cassandra. ***Conclusion:*** It is concluded that AML is a promising language for logical modeling in column-family NoSQL databases, proving effective in defining aggregates and DDD-based constructs.

Keywords: NoSQL; Logical Modeling; DDD; AML; Aggregates; Column families Databases; Apache Cassandra.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

Sigla	Significado
SQL	*Structured Query Language*
BD	Banco de Dados
TI	Tecnologia da Informação
SI	Sistemas de Informação
AML	*Aggregate Modeling Language*
DDD	*Domain-Driven Design*
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
IDC	*International Data Corporation*
BASE	*Basic Availability, Soft-state, Eventually consistent*
ORM	Mapeamento Objeto-Relacional
BOTE	*Back-Of-The-Envelope*
CQRS	*Command Query Responsibility Segregation*
DSML	*Domain Specific Modeling Language*

UML	*Unified Modeling Language*
DER	Diagrama Entidade-Relacionamento
EER	Entidade-Relacionamento Estendido

1. INTRODUÇÃO

1.1. Contexto

Os bancos de dados (BD) *NoSQL* surgiram como uma alternativa ao modelo relacional no processamento de *Big Data*. Com o avanço da *web*, aplicações que lidam com milhões de usuários passam a exigir disponibilidade e escalabilidade para processar *terabytes* e *petabytes* de dados não estruturados, como ocorre em redes sociais. Nesse contexto, os BDs *NoSQL* foram projetados para atender volumes massivos de dados estruturados e não estruturados, garantindo disponibilidade e tolerância à partição (LÓSCIO, 2011) (BREWER, 2012).

Dentre as diferentes categorias de BDs *NoSQL*, os BDs orientados à família de colunas destacam-se pelo seu desempenho no acesso aos dados quando comparados a BDs relacionais. Esse desempenho é resultado da abordagem desnormalizada de armazenamento, que agrupa logicamente estruturas esparsas por meio de uma chave única de identificação (LIMA, 2016). O Apache Cassandra é um dos BDs orientados à família de colunas mais utilizados que ocupa posição de destaque no ranking da *DBEngines* (DB-ENGINES, 2025). Sua popularidade deve-se a sua arquitetura distribuída e a sua eficiência na leitura e escrita em *Big Data* (CHEBOTKO, 2015). Por isso, a popularização desses BDs destaca pesquisas nesta área de Tecnologia da Informação (TI), especialmente no atual cenário de *Big Data*.

A implementação de BDs *NoSQL* apresenta alguns desafios, sendo a modelagem lógica um dos principais. A ausência de modelo bem definido é considerada uma das principais causas de falhas em Sistemas de Informação (SI) (BATINI; CERI; NAVATHE, 1991). No Projeto de banco de dados, a modelagem lógica é uma das etapas da modelagem direcionada à forma de armazenamento do BD alvo, com objetivo de criar modelos mais eficientes (HEUSER, 2009). Por isso que na modelagem lógica de BD *NoSQL* orientados a família de colunas, como no Apache Cassandra, os maiores desafios são as diferenças em relação às abordagens tradicionais de modelagem de dados (CHEBOTKO, 2015). Com a crescente adoção desses BDs, destaca-se a demanda por práticas de modelagem lógica que definam de forma mais expressiva como os dados serão persistidos no BD alvo (POFFO, 2016).

1.2. Motivação e Justificativa

Explorar os desafios de modelagem em BDs *NoSQL* orientados à família de colunas representa uma oportunidade para solucionar problemas neste cenário. À medida que o Apache Cassandra destaca-se como uma alternativa de armazenamento de dados distribuída, torna-se fundamental o uso de linguagens de modelagem que considerem suas particularidades e auxiliem no processo de definição lógica dos dados. Nesse contexto, destaca-se a *Aggregate Modeling Language* (AML), por se basear no conceito de agregados do *Domain-Driven Design* (DDD) (MONTEIRO, 2023), o qual, segundo Sadalage e Fowler, corresponde bem ao funcionamento da maioria dos BDs *NoSQL* (SADALAGE; FOWLER, 2013). No entanto, ainda faz-se necessário avaliar, na prática, a aplicabilidade da AML em BDs *NoSQL* orientados à família de colunas, especialmente o Apache Cassandra. Por isso, este trabalho se propõe a avaliar se a AML é capaz de representar as restrições, limitações e características do Apache Cassandra, bem como investigar se os construtores da AML são aplicáveis nesse tipo de BD.

1.3. Objetivos

1.3.1. Objetivo Geral

Demonstrar, por meio de um experimento prático, a viabilidade da modelagem lógica de BDs *NoSQL* orientados à família de colunas utilizando a AML, analisando se ela é capaz de representar as características, limitações e particularidades do Apache Cassandra.

1.3.1. Objetivos Específicos

(I) Avaliar se a AML é capaz de representar os principais conceitos, restrições e características do Apache Cassandra.

(II) Verificar a completude dos construtores da AML no Apache Cassandra: Aplicabilidade dos construtores da AML no Apache Cassandra.

(III) Explorar diferentes abordagens de modelagem lógica com AML sobre um mesmo modelo conceitual, utilizando variações estruturais para validar a completude da linguagem no Apache Cassandra.

(IV) Realizar um experimento prático com múltiplas abordagens de modelagem lógica aplicadas no Apache Cassandra, analisando suas implicações em termos de consistência, disponibilidade, tolerância a partições,

desempenho e escalabilidade com base nas boas práticas de modelagem de agregados que se alinham às diretrizes recomendadas pela equipe de desenvolvedores do Apache Cassandra.

2. REVISÃO DE LITERATURA

Neste capítulo é abordado os principais conceitos presentes neste trabalho. A seção 2.1 descreve os problemas na utilização de BD relacionais no cenário atual. Na seção 2.2, é detalhado os conceitos fundamentais de BD *NoSQL*. Já na seção 2.3, é apresentado o Apache Cassandra. Na seção 2.4, é abordado a importância do projeto de BD e suas etapas para utilização em organizações. Na seção 2.5, é explicado o conceito de agregados e a sua importância. E por fim, na seção 2.6 é apresentado o conceito e os construtores da linguagem de modelagem AML.

2.1. As Problemáticas dos Bancos de Dados Relacionais Tradicionais Centralizados

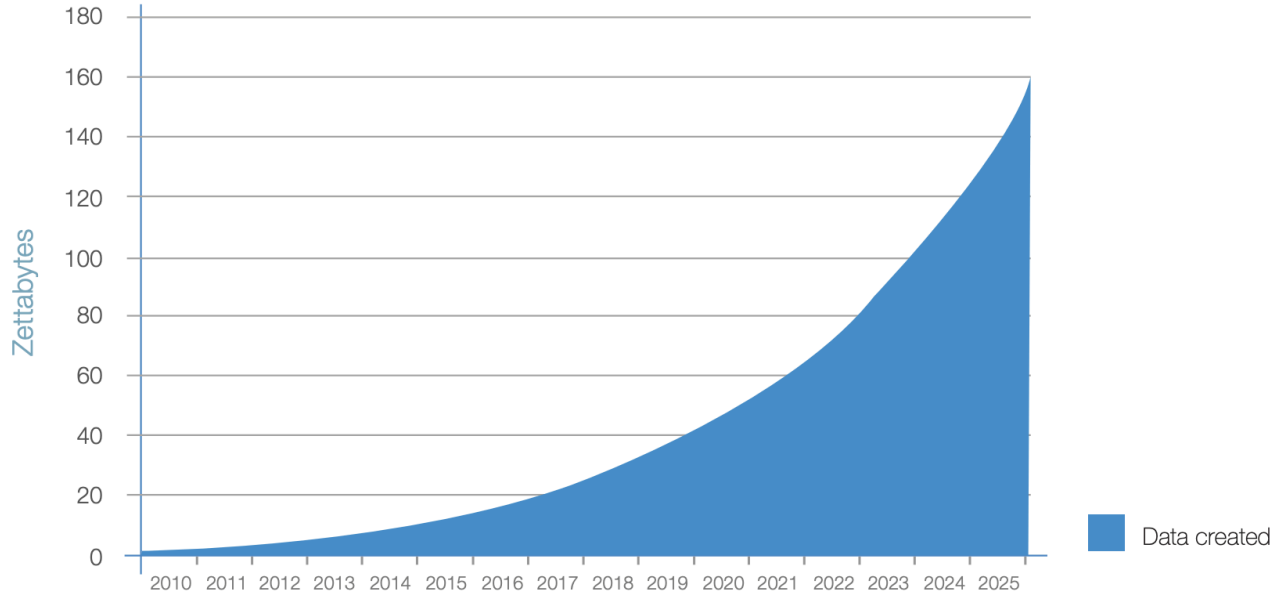
Os BDs relacionais dominaram o cenário do armazenamento de dados na internet, sendo recomendados para a maioria das aplicações. O *ranking* do DB-Engines confirma essa popularidade até os dias atuais, mostrando que, entre os dez BDs mais bem posicionados, seis são relacionais (DB-ENGINES, 2025). Esse destaque deve-se à interface padronizada e à capacidade de garantir um armazenamento consistente e disponível dos dados. Todos os principais BDs relacionais utilizam dialetos (ex.: *SELECT*, *UPDATE*, *INSERT*, *DELETE*) do *Structured Query Language* (SQL), que facilita a sua adoção como plataforma compartilhada, conhecidas como BDs de integração, que permitem que diferentes aplicações na organização utilizem-o de forma centralizada para integração (SADALAGE; FOWLER, 2013). Além dessa uniformidade, os BDs relacionais foram projetados para lidar com interação simultânea de usuários por meio do controle transacional que aplicam as características do acrônimo ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Portanto, essas propriedades asseguram operações de armazenamento seguras e confiáveis (LÓSCIO, 2011).

Com a evolução da internet e o crescimento da conectividade global, tornou-se mais relevante a demanda por processamento em tempo real de grandes volumes de dados não estruturados. A popularização de serviços como redes sociais, serviços de *streaming* e sistemas de análise de *logs* evidenciou a necessidade de estratégias mais eficientes para tratamento e armazenamento de dados complexos, conhecidos como *Big Data*. Essa complexidade é caracterizada por cinco dimensões conhecidas como os 5Vs que são (ANURADHA, 2015):

1. Volume

O volume refere-se à grande quantidade de dados gerados atualmente. Segundo estimativas do *Data Age* de 2025 do *International Data Corporation* (IDC) a esfera global de dados superará os 163 *zetabytes* (ZB), representando um aumento de dez vezes em relação à quantidade de dados gerada em 2016 (REINSEL; GANTZ; RYDNING, 2017). A Figura 1 ilustra esse crescimento ao longo dos anos, segundo o IDC.

Figura 1 - Crescimento da esfera global de dados segundo IDC



Fonte - REINSEL GANTZ RYDNING (2017).

2. Velocidade

A velocidade relaciona-se com a rapidez com que os dados são gerados e com que eles se movem (ANURADHA, 2015). A popularidade dos *smartphones*, com milhões de dispositivos conectados, e pelos dispositivos *IoT*, como sensores, impulsionou uma produção contínua e acelerada de informações. A Figura 2 apresenta o infográfico do *Data Never Sleeps* 2023 da DOMO, que demonstra uma estimativa da quantidade de dados gerados atualmente em diferentes aplicações no intervalo de 60 segundos (DOMO, 2023).

Figura 2 - Infográfico de geração de dados em 60 segundos - *Data Never Sleeps* da DOMO

Fonte - DOMO (2023).

3. Variedade

A variedade diz respeito à diversidade de formatos e origem dos dados, abrangendo desde informações estruturadas (ex.: BDs relacionais) até não estruturadas (ex.: imagens, vídeos, áudios) e semiestruturadas (ex.: *JSON*, *XML*) ANURADHA, 2015.

4. Veracidade

A veracidade refere-se à qualidade e confiabilidade dos dados. O processamento de grande volume de dados com formatos e origens distintas pode conter imprecisões, tornando essencial a implementação de técnicas que garantam a integridade e a confiabilidade dos dados (ANURADHA,

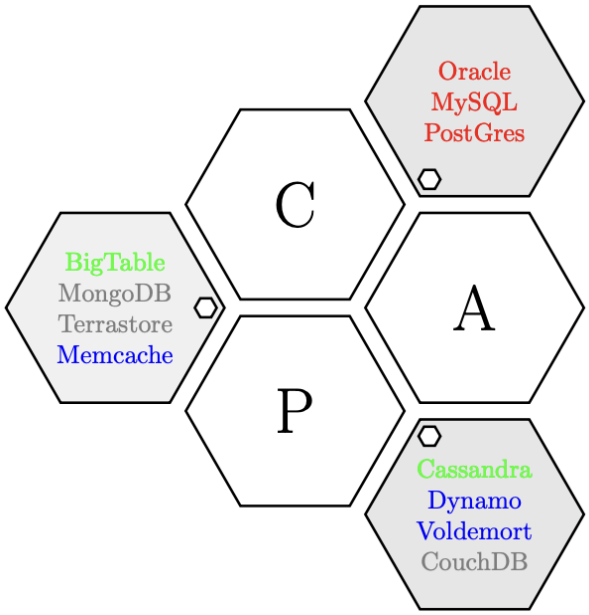
2015). Por exemplo, um sensor de temperatura de *IoT* apresenta um *bug* na coleta de dados aumentando em 10º C, então os dados a partir deste devem ser descartados ou tratados para utilização.

5. Valor

O valor relaciona-se à capacidade de transformar os dados brutos em valor para o negócio, ou seja, transformá-los em algo útil para a tomada de decisões estratégicas e geração de *insights* para os negócios (ANURADHA, 2015). Por exemplo, os dados de *logs* gerados a partir da utilização de um sistema que pode ser transformado para gerar *insights* de comportamento do usuário.

Para lidar com esse cenário de aumento do volume e complexidade dos dados, é necessário ampliar a capacidade computacional por meio da escalabilidade vertical ou horizontal. A escalabilidade vertical consiste em adquirir recursos com maior capacidade computacional, porém há um limite físico e um aumento exponencial de custos. Como alternativa, a escalabilidade horizontal distribui a carga de trabalho entre múltiplos recursos menores organizados em um *cluster* que torna o sistema resiliente e acessível. No entanto, os BD relacionais tradicionais centralizados não foram projetados para operar de forma distribuída (SADALAGE; FOWLER, 2013). Essa limitação é explicada pelo Teorema de CAP indicado na Figura 3, que estabelece que um sistema de armazenamento compartilhado de dados distribuído só pode garantir totalmente duas propriedades simultaneamente entre: consistência (representado como C do *Consistency* na Figura 3), disponibilidade (representado como A do *Availability* na Figura 3) e tolerância ao particionamento (representado como P com *Partition Tolerance* na Figura 3). Note-se, por exemplo, que na Figura 3 os BD relacionais tradicionais centralizados como *Oracle*, *MySQL* e *Postgres* priorizam consistência (C) e disponibilidade (A) em detrimento a tolerância à partição sem comprometer o desempenho ou a integridade dos dados (SOUSA, 2010) (BREWER, 2012) (POFFO, 2016).

Figura 3 - Teorema CAP com exemplos de BDs para cada combinação de duas características



Fonte - POFFO (2016).

O avanço do *Big Data* evidenciou a necessidade de um modelo alternativo de armazenamento diante das limitações dos BDs relacionais tradicionais centralizados. Com isso, levou ao surgimento do movimento *NoSQL*, abordagem que oferece maior flexibilidade, escalabilidade horizontal e eficiência no armazenamento e processamento de grandes volumes de dados diversos, atendendo às novas demandas tecnológicas (SADALAGE; FOWLER, 2013). Por exemplo, como ilustrado na Figura 3, os BDs *NoSQL* priorizam a tolerância à partição (P) e permitem um modelo de consistência flexível conforme o cenário. Alguns BDs priorizam a consistência (C), como BigTable, MongoDB, Terrastore e Memcache, enquanto outros priorizam a disponibilidade (A), como Cassandra, Dynamo, Voldemort e CouchDB (POFFO, 2016).

2.2. Bancos de Dados *NoSQL*

Segundo Sadalage e Fowler, *NoSQL* é um neologismo acidental que se popularizou como uma solução alternativa de armazenamento aos BDs relacionais tradicionais centralizados (SADALAGE; FOWLER, 2013). O termo foi difundido por Eric Evans, que o definiu como "uma alternativa de armazenamento para um problema que você não consegue resolver com bancos

relacionais" (ALBUQUERQUE, 2017) (STRAUCH; KRIHA, 2011). Embora não haja uma definição clara para o termo *NoSQL*, há um consenso sobre a existência de algumas características comuns entre esses sistemas que são: modelo não relacional, execução em *clusters*, *open source* e ausência de esquema definido (ALBUQUERQUE, 2017) (SADALAGE; FOWLER, 2013).

A arquitetura do BD *NoSQL é* projetada para operar de forma distribuída com garantia de escalabilidade e tolerância a falhas, o que os tornam ideais para aplicações lidam com *Big Data* (SADALAGE; FOWLER, 2013). Diferentemente dos BDs relacionais, que priorizam a consistência por meio do modelo ACID, os BDs *NoSQL* adotam um modelo mais flexível conhecido como BASE (*Basic Availability, Soft-state, Eventually consistent*). Este modelo garante que os BDs *NoSQL* estão quase sempre disponíveis, não estão continuamente consistentes e as atualizações são propagadas eventualmente. (ALBUQUERQUE, 2017) (STRAUCH; KRIHA, 2011). Para isso, seus dados são distribuídos entre as nós (máquinas) do *cluster*, até de forma duplicada, para diminuir o impacto da escalabilidade horizontal e manter a disponibilidade (ALBUQUERQUE, 2017) (STRAUCH; KRIHA, 2011). No entanto, conforme estabelece o Teorema de CAP, a responsabilidade pela consistência dos dados recai sobre a aplicação, já que os BDs *NoSQL* geralmente acabam priorizando a disponibilidade e a tolerância à partição, em detrimento da consistência (BREWER, 2012). Por exemplo, em BDs distribuídos como o Apache Cassandra, onde os dados são distribuídos e replicados de acordo com a chave de partição entre múltiplos nós do *cluster*. Embora esse modelo ofereça escalabilidade e disponibilidade, algumas leituras podem temporariamente recuperar dados desatualizados devido ao atraso de propagação das atualizações entre os nós.

Outro fator que impulsiona a adoção dos BDs *NoSQL* é o aumento da produtividade no desenvolvimento de aplicações devido a utilização de estruturas parecidas com a estrutura de dados na memória. Pois, os BDs *NoSQL* aplicam estruturas ricas e, geralmente, sem esquema pré-definido nas suas *APIs* que se aproximam das linguagens de programação, o que permite maior flexibilidade na modelagem de dados e reduz a necessidade de *frameworks* de mapeamento objeto-relacional (*ORM*). Um exemplo disso é o MongoDB, que é um BD orientados a documentos que armazena os dados no formato *JSON* que é um formato bastante utilizado nas aplicações *web* atualmente.

Segundo Sadalage e Fowler, a maior contribuição do movimento *NoSQL* nas organizações é a

persistência poliglota, que consiste no uso de diferentes tipos de armazenamento de dados em cenários distintos (SADALAGE; FOWLER, 2013). Dessa forma, ao escolher um tipo de BD para uma aplicação, faz-se necessário entender a natureza dos dados e como ele é manipulado para que seja escolhido o BD que atende às necessidades da aplicação. Nos BDs *NoSQL*, existem 4 categorias que tratam diferentes tipos de dados, sendo elas:

- **Chave/valor**: Modelo de armazenamento baseado em uma tabela *hash*, em que cada chave primária é associada a um valor. O valor armazenado pode ter qualquer formato (ex.: *blob*, texto, *JSON*, *XML*), sendo responsabilidade da aplicação interpretar seu conteúdo. Como a busca ocorre exclusivamente pela chave, esses BDs são otimizados para leituras e gravações e permitem escalabilidade horizontal. São utilizados em cenários que a *performance* é prioritária, como *caches* de sessão, gerenciamento de *tokens* e filas de mensagens. Exemplos desse modelo incluem Redis, Amazon DynamoDB e Riak (SADALAGE; FOWLER, 2013).
- **Orientados a documentos**: BDs projetados para armazenar documentos estruturados em coleções, utilizando formatos como *JSON*, *BJSON* e *XML*. Esses documentos são estruturas de dados na forma de árvores hierárquicas e autodescritivas com mapas, coleções e valores escalares. A ausência de um esquema fixo proporciona flexibilidade, permitindo armazenar documentos semelhantes e realizar consultas dinâmicas. Esses BDs são utilizados em aplicações *web* que manipulam documentos (ex.: *JSON*) e em sistemas que exigem agregação de dados hierárquicos, permitem consistência eventual e precisam realizar consultas dinâmicas. Exemplos incluem MongoDB, CouchDB e RavenDB (SADALAGE; FOWLER, 2013).
- **Colunares ou orientados à família de colunas**: Modelo de armazenamento baseado em família de colunas, organizadas e acessadas por meio de uma chave primária. Essas famílias agrupam dados relacionados que são armazenados fisicamente próximos para otimizar o desempenho das consultas. Esse tipo de BD permite um esquema flexível, possibilitando a adição de novas colunas sem necessidade de *downtime*. Apresentam escalabilidade horizontal com otimização para operações tanto de escrita quanto de consulta. Esse modelo é utilizado em aplicações que lidam com grandes volumes de dados, como análises de séries temporais, monitoramento de *logs* e processamento de eventos em *real time*. Este modelo de dados é o foco deste trabalho e será detalhado nos próximos capítulos. Exemplos de BD que adotam essa abordagem incluem Apache Cassandra (foco deste trabalho), ScyllaDB e Apache HBase (SADALAGE; FOWLER,

2013).

- **Orientado a grafos**: Modelo de BD projetado para armazenar entidades (nós) e seus relacionamentos (arestas), ambos podendo conter atributos. Os nós representam os objetos armazenados, enquanto as arestas definem os relacionamentos entre eles, possuindo direção e significado. Esse modelo é ideal para aplicações que priorizam as análises de relações, como redes sociais, mecanismos de recomendação, detecção de fraudes e análise de grafos em *Big data*. Exemplos incluem Neo4j, ArangoDB e Amazon Neptune (SADALAGE; FOWLER, 2013).

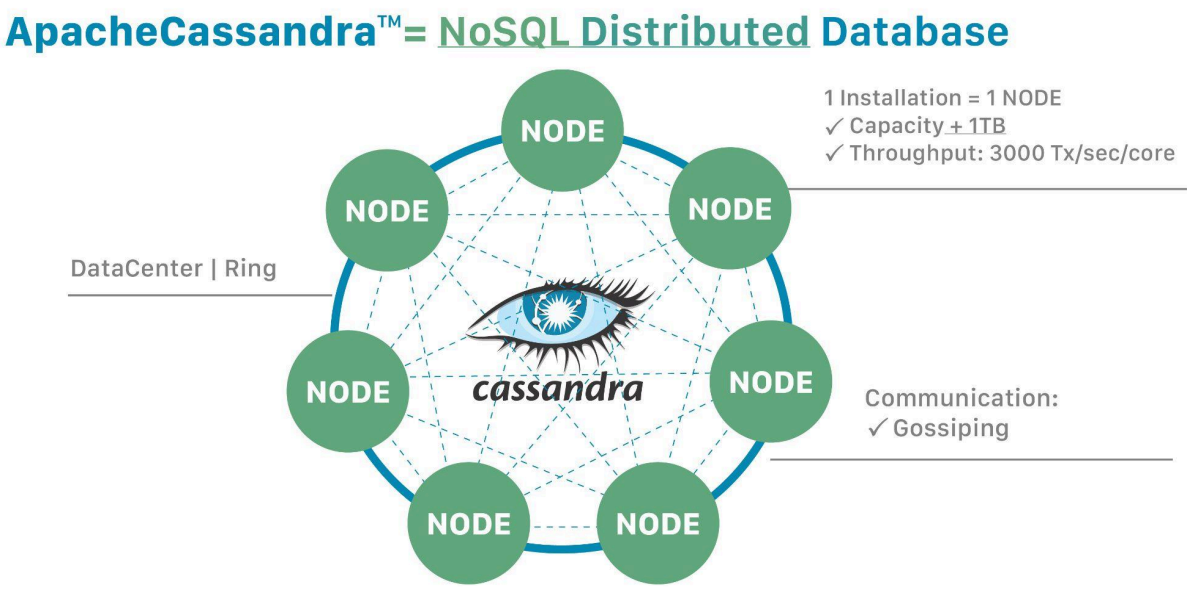
2.3. Apache Cassandra

O Apache Cassandra é um BD *NoSQL* do tipo família de colunas, amplamente utilizado em aplicações de *Big Data* devido à sua arquitetura distribuída, escalabilidade horizontal e tolerância a falhas (CASSANDRA, 2025). Desenvolvido por Avinash Lakshman, coautor do Amazon DynamoDB, e Prashant Malik no Facebook, o Cassandra foi criado para gerenciar a pesquisa da caixa de entrada da rede social, oferecendo uma solução escalável e eficiente para lidar com grandes volumes de dados gerados diariamente. Seu nome faz referência à profetisa Trojan Cassandra, da mitologia grega, simbolizando sua capacidade de "prever" e gerenciar grandes quantidades de dados de forma eficiente (WIKIPÉDIA, 2025). Atualmente, ocupa a 12ª posição no *ranking* de popularidade do *DB-Engines* e destaca-se por sua arquitetura descentralizada, que possibilita leituras e escritas eficientes, suportando milhões de transações por segundo, além de oferecer uma definição flexível de esquemas e uma linguagem de consulta intuitiva, facilitando sua adoção em diversos cenários como plataformas de *streaming*, financeiros com armazenamento de eventos e dispositivos de *IoT* (DB-ENGINES, 2025) (DATASTAX, 2025).

A principal característica do Apache Cassandra é sua arquitetura distribuída, baseada em *clusters* ou "*rings*", compostos por múltiplos nós (*nodes*) que podem ser adicionados dinamicamente, sem interrupção no serviço, como demonstrado na Figura 4. Por exemplo, se um *cluster* contém três nós e um conjunto de dados é inserido, esses dados são particionados e armazenados em diferentes nós para balanceamento da carga e redundância. Essa capacidade permite que as aplicações escalem horizontalmente conforme a demanda. Os nós comunicam-se

entre si de forma *peer-to-peer* por meio do protocolo de comunicação *gossip*, que propaga informações sobre o estado do *cluster*, garantindo que cada nó tenha conhecimento sobre a localização e o status dos demais. Esse mecanismo funciona de maneira semelhante à propagação de rumores em um grupo de pessoas: quando um nó recebe uma atualização sobre o estado do *cluster*, ele compartilha essa informação com os nós vizinhos, que por sua vez retransmitem para outros, espalhando a informação de forma descentralizada. Por exemplo, se um nó do *cluster* falha ou um nó é adicionado, os demais tomam conhecimento dessas mudanças gradualmente, sem depender de um servidor central. No Cassandra, todos os nós desempenham o papel de coordenador, distribuindo as requisições de forma descentralizada. Essa estrutura elimina pontos únicos de falha, tornando o sistema mais resiliente e disponível, mesmo em cenários de falha parcial da infraestrutura (DATASTAX, 2025) (CASSANDRA, 2025).

Figura 4 -Arquitetura Apache Cassandra

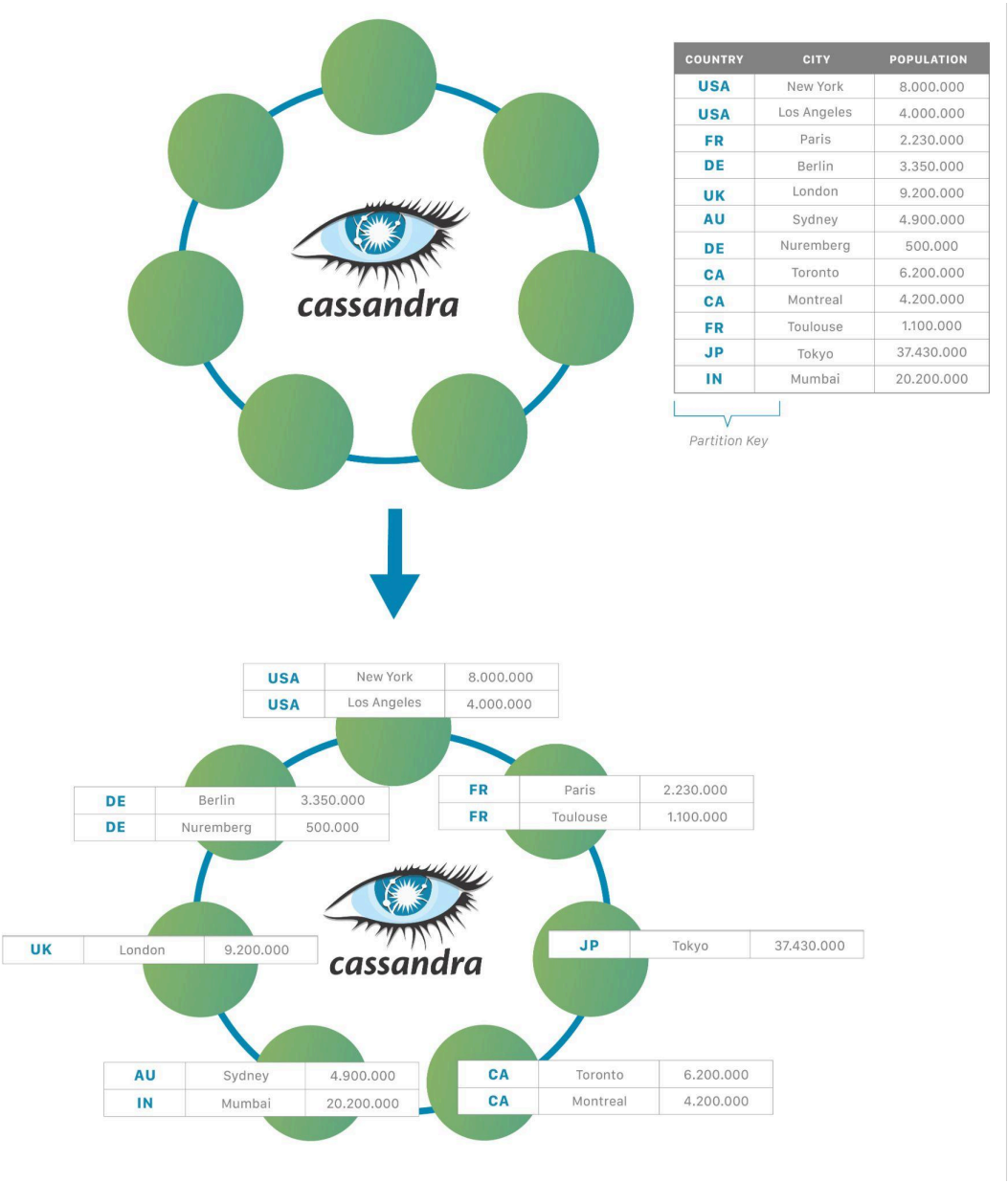


Fonte - CASSANDRA (2025).

Para garantir desempenho no acesso aos dados, o Cassandra distribui automaticamente as informações no *cluster* por meio de partições, utilizando a chave de partição como critério de distribuição. Conforme ilustrado na Figura 5, cada nó possui um conjunto de *tokens*, que determinam quais partições devem ser armazenadas. Quando os dados são manipulados, uma função *hash,* conhecida como *Partitioner,* é aplicada à chave de partição para calcular o *token* correspondente e identificar o nó responsável pelo armazenamento, como ilustrado na Figura 6.

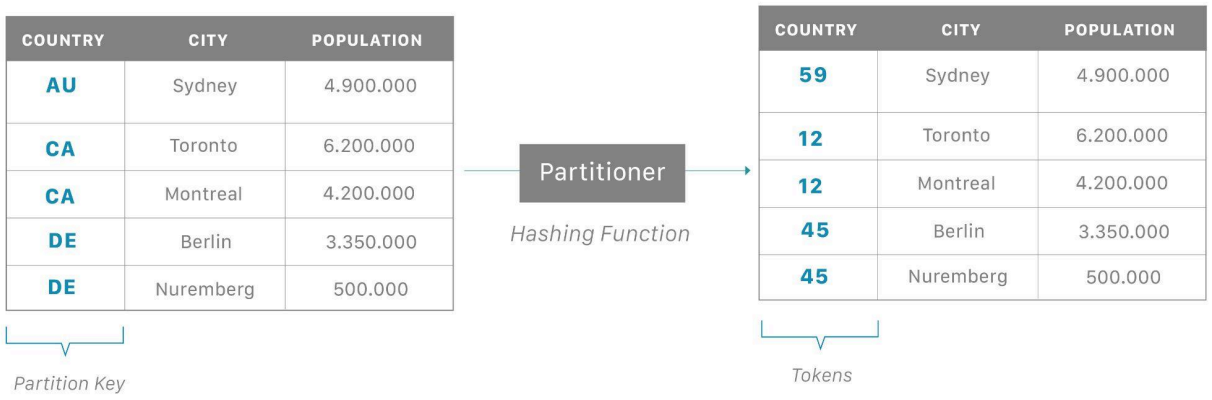
Este processo pode ser iniciado a partir de qualquer nó do *cluster*, o que contribui para a eficiência e flexibilidade do Cassandra. Por isso, a escolha da chave de partição é um fator essencial na modelagem das tabelas, pois influencia diretamente a distribuição equilibrada dos dados e desempenho das consultas. Por exemplo, em um sistema de *e-commerce*, definir a chave de partição como o identificador do usuário pode ser uma escolha eficiente para armazenar os pedidos do cliente de forma distribuída e garantir consultas rápidas. No entanto, a escolha inadequada da chave de partição no Apache Cassandra pode impactar em *hotspots*, quando uma chave de partição possui baixa cardinalidade, concentrando dados em um único nó. Um exemplo disso seria definir a chave de partição como o nome do país em um sistema de registro de transações, pois países teriam um número de registros que sobrecarregariam nós específicos. Por outro lado, se a chave de partição for muito fragmentada, como o uso de um identificador único de pedido em um sistema, onde há necessidade de recuperar múltiplos pedidos de um cliente, a consulta precisará acessar vários nós do *cluster* para obter os dados, resultando em perda de *performance*. Por isso, para evitar essas problemáticas, é essencial definir a chave de partição que garanta uma distribuição uniforme e agrupada, alinhada com o padrão de acesso das consultas e evitando *hotspots* quanto a fragmentação de partições (CASSANDRA, 2025).

Figura 5 - Demonstração da distribuição dos dados no *cluster* do Apache Cassandra.



Fonte - CASSANDRA (2025).

Figura 6 - Ilustração do funcionamento da função *Partitioner* do Apache Cassandra



Fonte - CASSANDRA (2025).

O Cassandra utiliza a linguagem *Cassandra Query Language* (*CQL*) para criação, modificação, exclusão e consulta de dados, com uma sintaxe semelhante ao SQL. Seus dados são organizados nas seguintes estruturas CASSANDRA (2025):

- ***Keyspace***: Funciona como um *namespace* que agrupa tabelas relacionadas que compartilham as mesmas configurações de replicação que é determinado por um parâmetro que indica a quantidade de cópias de dados distribuídas no *cluster*.
    - Exemplo: Armazenamento de registros de usuário que os dados serão replicados automaticamente entre diferentes regiões para garantir disponibilidade.
- **Tabela**: Estrutura de organização de dados em linhas e colunas, em que cada coluna segue um tipo de dado definido. As tabelas permitem a adição de novas colunas sem necessidade de *downtime*.
    - Exemplo: Em um sistema de *e-commerce*, uma tabela de *pedidos* pode ter colunas para o *identificador do cliente*, *data da compra* e *status do pedido*, sendo possível adicionar uma nova coluna para o *método de pagamento* sem interromper o funcionamento do BD.
- **Partição**: Define em qual nó do *cluster* uma linha será armazenada, sendo representada pela chave de partição, que faz parte da chave primária. Todas as consultas devem incluir essa

chave, a menos que um índice seja utilizado.

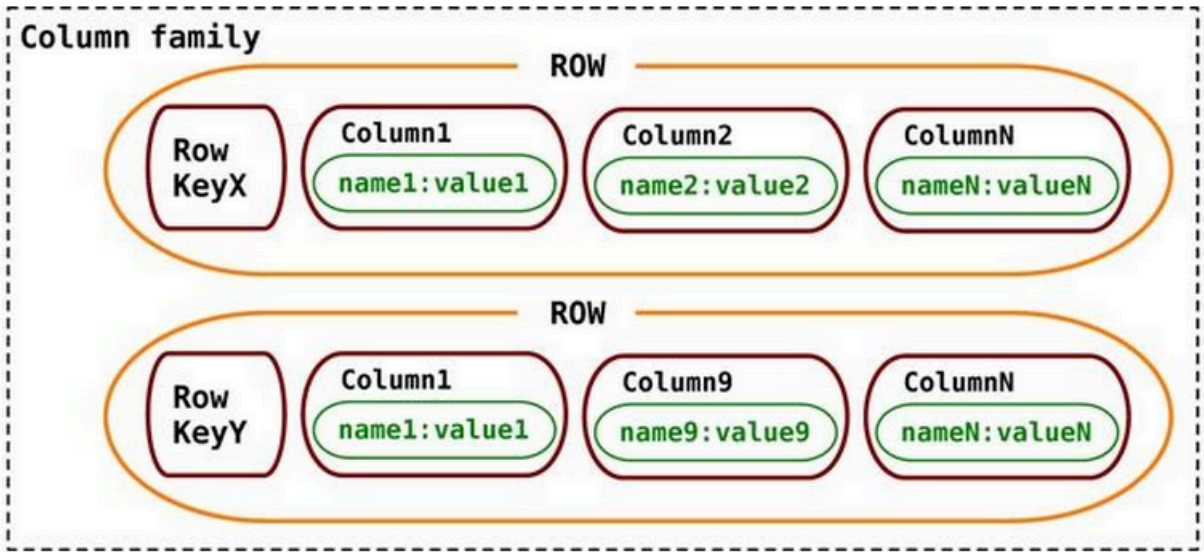
- Exemplo: Em um sistema de *streaming*, os registros de *visualizações* podem ser particionados pelo *identificador do usuário*, garantindo que os dados de um mesmo usuário fiquem agrupados para facilitar o acesso e otimizar consultas.

- **Linha**: Conjunto de colunas (ou família de colunas) identificadas e agrupadas por uma chave primária, que são armazenadas em ordem pela chave primária. Dentro de uma mesma partição, os registros são ordenados de acordo com as chaves de ordenação. Conforme ilustrado na Figura 7;
  - Exemplo: Em uma tabela que armazena *transações financeiras*, uma linha pode conter colunas como *valor da transação*, *data* e *categoria do gasto*, com chave primária composta pelo *identificador do usuário* e a *data da transação*. O *identificador do usuário* é a chave de partição, que particiona os registros entre os nós, e a *data de transação*, sendo a chave de ordenação que ordena os registros dentro da mesma partição.
- **Coluna**: Unidade básica de armazenamento de dados que representa um valor tipado pertencente a uma linha.
  - Exemplo: Em um sistema de suporte, uma coluna pode armazenar o *status do chamado*, como aberto, em andamento ou resolvido.
- **Chave Primária**: Define a identidade única de uma linha na tabela e é composta pela chave de partição (*partition key*) e, opcionalmente, uma ou mais colunas de ordenação (*clustering keys*). No Apache Cassandra, a ordem das colunas na declaração da chave primária importa: a primeira coluna sempre representa a chave de partição, enquanto as demais, se existirem, compõem a chave de ordenação. Essa estrutura influencia diretamente a distribuição dos dados no cluster e a ordenação física dos registros dentro de uma mesma partição.
  - Exemplo: Em um sistema de reservas de hotéis, a chave primária pode ser composta pelo *identificador do hotel* (chave de partição) e a *data da reserva* (chave de ordenação), garantindo que todas as reservas do mesmo hotel fiquem agrupadas.
- **Chave de Ordenação**: Define a ordenação das linhas dentro da partição, otimizando buscas em consultas;

- Exemplo: Em um fórum, os comentários podem ser armazenados com uma chave de ordenação baseada na *data de publicação*, permitindo a exibição em ordem cronológica.

- ***Materialized View* (MV):** São tabelas construídas a partir de dados de outra tabela base com outra chave primária e, possivelmente, novas colunas. Alterações de dados na tabela original são replicadas automaticamente nas MVs. Pode ser utilizado para criar tabelas específicas para consultas performáticas. Porém, seu uso em produção não é recomendado, pois ainda é experimental devido a problemas de sincronização de dados a partir das tabelas base (*ticket* no projeto do Jira do Apache Cassandra: *CASSANDRA-10346*).
  - Exemplo: Uma tabela de *sistema de pedidos* que tem como chave primária o *identificador do pedido*. Uma MV pode ser utilizada para gerar uma tabela otimizada para consulta utilizando uma outra chave primária: o *identificador do usuário* e a *data do pedido*.

Figura 7 - Modelo de dados do BD *NoSQL* com família de colunas



Fonte - FERREIRA (2019).

O CQL é uma linguagem tipada que oferece suporte a diversos tipos de dados, incluindo tipos nativos, *User-Defined Types* (UDTs), tuplas e tipos personalizados. A seguir, um detalhamento de cada categoria CASSANDRA (2025):

- **Tipos nativos (*Native Types*)**: O CQL oferece suporte a diversos tipos nativos, como *ascii,*

bigint, blob, boolean, counter, date, decimal, double, duration, float, inet, int, smallint, text, time, timestamp, timuuid, tinyint, uuid, varchar, varint e *vector*. A Tabela 1 detalha cada tipo nativo, suas constantes suportadas e suas respectivas descrições.

- **Tipos de coleção (*Collection Types*)**: Permitem o armazenamento desnormalizado de pequenos conjuntos de dados dentro de uma única coluna, informações como telefones e e-mails de um cliente. No entanto, coleções não devem ser usadas para armazenar dados que crescem indefinidamente, como mensagens enviadas por um usuário ou *logs* de um sensor, pois podem comprometer a escalabilidade do Cassandra. As coleções não possuem indexação interna, ou seja, ao buscar um item na coleção, o Cassandra retorna a coleção inteira ao invés de um único item. Além disso, uma coleção pode ser definida como *frozen* que indica que todos os componentes da coleção são serializados em um valor único como um *blob*. Com isso, o Apache Cassandra subscreve o valor inteiro ao invés de atualizar campos individuais como nos tipos *non-frozen*. O *CQL* oferece 3 tipos principais de coleções:
    - ***Maps*:** Estrutura que armazena pares chave-valor ordenados por chave. Cada chave é única e os valores são acessados a partir dela.
        - Exemplo: *Endereços*, onde a chave é o *tipo de endereço* e o valor é o próprio endereço, como *{ "casa": "Rua A, 123", "trabalho": "Av. B, 456" }*.
    - ***Lists*:** Uma coleção ordenada de valores que permite valores duplicados e mantém a ordenação de inserção, ou seja, os primeiros elementos adicionados são retornados primeiro.
        - Exemplo: *Telefones* de um cliente, garantindo que não haja repetições, como *{ "+55 11 99999-0000", "+55 21 98888-1111" }*.
    - ***Sets*:** Conjunto ordenado de valores únicos, ou seja, não permite valores duplicados e armazenados em ordem crescente.
        - Exemplo: Ações de um usuário em um aplicativo, garantindo que não haja repetição, como *["home", "produtos", "checkout", "confirmação"]*.
- ***User-Defined Types* (UDTs)**: Permitem a criação de tipos personalizados, que é composto por um nome (que indica seu uso nas tabelas) e as propriedades com tipos definidos. Essas propriedades podem incluir outros UDTs, permitindo estruturas complexas. Os UDTs são

úteis para armazenar dados estruturados diretamente em colunas.

  - Exemplo: *Local de entrega* com atributos como *{ rua: text, cidade: text, estado: text, cep: int }*.

- **Tipos de tuplas (*Tuple Types*)**: Estrutura que agrupa múltiplos valores em uma única coluna, preservando a ordem de inserção. Ao contrário dos UDTs, as tuplas não nomeiam os campos internos, sendo referenciadas apenas por sua posição.
  - Exemplo: Armazenar coordenadas geográficas como *(-23.5505, -46.6333)*.
- **Tipos personalizados (*Custom Types*)**: Permitem a criação de tipos específicos definidos pelo usuário, que referenciam classes Java personalizadas carregadores pelo Cassandra. Porém, seu uso é desencorajado devido à complexidade de implementação e manutenção. Em geral, UDTs são suficientes para atender às necessidades de personalização.

Tabela 1 – Detalhamento dos tipos do CQL

Tipo	**Constante suportada**	**Descrição**
ascii	string	*String* de caracteres *ASCII*.
bigint	integer	*Long* com sinal de 64 *bits*.
blob	blob	*Bytes* arbitrários (sem validação).
boolean	boolean	Pode ser verdadeiro (*true*) ou falso (*false*).
counter	integer	Coluna de contador (valor inteiro com sinal de 64 *bits*) que permite duas operações automáticas: incremento e decremento. Ou seja, ao inserir um dado, será incrementado ou diminuído de acordo com a configuração da coluna. Esse tipo de dado não pode ser alterado por *update* pelo usuário.
date	integer, string	Uma data que é codificada como um inteiro de 32 *bits* representando o número de dias desde o *epoch time* (1 de janeiro de 1970).
decimal	integer, float	Decimal de precisão variável.

Tipo	Constante suportada	Descrição
double	integer float	Ponto flutuante de 64 bits *IEEE-754*.
duration	duration,	Uma duração com precisão de nanossegundos. É codificado em 3 números inteiros que o primeiro inteiro representa a quantidade de meses, segundo a quantidade de dias e o último a quantidade de nanosegundos.
float	integer, float	Ponto flutuante de 32 bits IEEE-754.
inet	string	Um endereço IP, seja *IPv4* (4 *bytes*) ou *IPv6* (16 *bytes*). Observe que não há uma constante *inet*, o endereço IP deve ser inserido como uma *string*.
int	integer	Inteiro com sinal de 32 *bits*.
smallint	integer	Inteiro com sinal de 16 *bits*.
text	string	*String* codificada em *UTF8*
time	integer, string	Um horário (sem valor de data correspondente) com precisão de nanossegundos. Os valores são codificados em inteiro 64 *bits* que representa os nanosegundos desde a meia noite.
timestamp	integer, string	Um carimbo de data e hora (data e tempo) com precisão de milissegundos. Os valores são codificados em inteiro 64 *bits* que representa os nanosegundos desde a *epoch time*.
timeuuid	uuid	*UUID* de versão 1, geralmente usado como um carimbo de data/hora “livre de conflitos” que pode ser usado como chave primária.
tinyint	integer	Inteiro com sinal de 8 *bits*.
uuid	uuid	Um *UUID* (de qualquer versão).
varchar	string	*String* codificada em *UTF8*.

Tipo	Constante suportada	Descrição
varint	integer	Inteiro de precisão arbitrária.
vector	float	Um *array* fixo de valores de ponto flutuante não nulos e achatados. Esse tipo de dado foi adicionado ao Cassandra 5.0 no *CASSANDRA-18504*.

Fonte - CASSANDRA (2025).

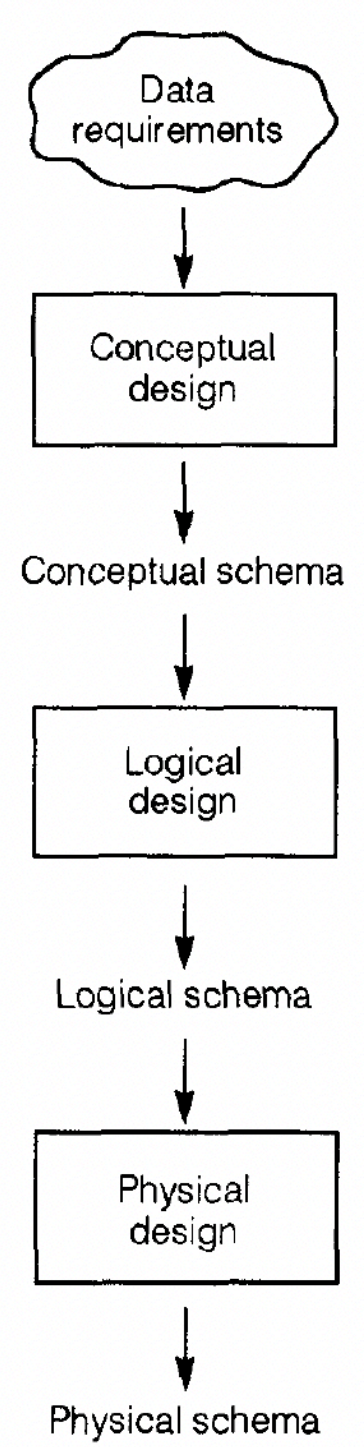
Uma outra funcionalidade importante é a definição da coluna *STATIC* que permite o armazenamento de um único valor compartilhado por todas as linhas da partição. Foi introduzida no Apache Cassandra 3.x que reduz a redundância de dados e economia de armazenamento, sendo ideal para armazenar informações que permanecem constantes dentro de uma partição CASSANDRA (2025). Por exemplo, em uma tabela *cliente*, em que cada partição representa um cliente, uma coluna estática pode ser utilizada para armazenar o *nome do cliente*. Dessa forma, o nome é armazenado apenas uma vez por partição, enquanto outras colunas, como *histórico de transações*, podem variar entre as linhas sem necessidade de repetição do nome em cada registro.

2.4. Projeto de Banco de Dados

A modelagem de dados é uma atividade fundamental no desenvolvimento de BD, pois sua ausência é considerada uma das principais causas de falha em SI. BDs sem uma estrutura de modelagem de dados no seu desenvolvimento tendem a ser ineficientes e inadequados para atender às necessidades das aplicações, além de apresentarem uma documentação limitada e uma manutenção custosa (BATINI;CERI;NAVATHE, 1991). Para estruturar um BD, utilizam-se linguagens de modelagem que podem ser gráficas ou textuais que representam formalmente como os dados são organizados no BD alvo. Essas representações são conhecidas como esquemas de BD, que descrevem os tipos de informações armazenadas e sua disposição. A modelagem envolve diferentes níveis de abstração, desde modelos mais abstratos, que descrevem regras de negócio, até modelos mais detalhados, que descrevem a organização interna dos dados, para atender a diferentes necessidades (HEUSER, 2009) (BATINI;CERI;NAVATHE, 1991). Por isso, o projeto de BD visa definir a estrutura do BD em 3 etapas que são representadas com seus parâmetros e resultados em ordem na Figura 8: *Design* Conceitual, Lógico e Físico:

- ***Design* conceitual:** Essa fase inicia com a especificação dos requisitos de negócio e resulta em um modelo abstrato, que descreve os dados e as informações que serão armazenadas no BD, sem considerar aspectos de implementação ou estrutura de armazenamento. Esse modelo, conhecido como modelo conceitual do BD, fornece uma descrição de alto nível da estrutura do BD, independente do SGBD alvo escolhido para implementação.
- ***Design* Lógico**: Essa fase utiliza o modelo conceitual como base para desenvolver o modelo lógico, que descreve a estrutura de armazenamento que será utilizada no BD de acordo com o tipo de dado a ser utilizado (relacional ou não relacional). Diferente do modelo conceitual, o modelo lógico já considera restrições técnicas, mas ainda não está vinculado a um SGBD específico, permitindo que um mesmo modelo lógico possa ser implementado em diferentes plataformas. Essa fase é o foco deste trabalho e será detalhado nos próximos capítulos.
- ***Design* Físico**: Nessa etapa, o modelo lógico é traduzido para um modelo físico, considerando as particularidades do SGBD escolhido. Essa fase descreve estruturas de armazenamento e índices, além de levar em consideração os métodos de acesso aos dados para desenvolver um modelo eficiente nas operações. Além de documentar a implementação, essa fase também permite ajustes para melhorar a *performance*, levando em conta fatores como particionamento, replicação e otimização de consultas.

Figura 8 - Etapas do Projeto de Banco de dados



Fonte - BATINI;CERI;NAVATHE (1991)

2.4.1. Design Lógico

O *design* lógico é uma das etapas mais importantes no desenvolvimento de um projeto de BD, pois define um modelo adaptado à estrutura dos dados do BD alvo a partir do modelo conceitual, garantindo que os requisitos de negócio sejam respeitados. No caso do Apache Cassandra, a modelagem precisa ser adaptada às particularidades de um BD distribuído e orientado a famílias de colunas. Diferente dos BDs relacionais, que utilizam tabelas normalizadas e relacionamento entre entidades, o Cassandra adota um modelo desnormalizado, no qual os dados são organizados de forma otimizada para leitura e escrita eficiente. O modelo lógico do Cassandra baseia-se na organização dos dados em famílias de colunas, em que cada linha contém um conjunto de colunas dinâmicas associadas a uma chave primária que consiste na chave de partição e, opcionalmente, uma ou mais chaves de ordenação (*clustering*). A escolha da chave de partição é um aspecto na modelagem lógica, pois influencia na distribuição dos dados entre os nós do *cluster* e o desempenho das consultas. Dessa forma, para garantir um balanceamento eficiente e evitar *hotspots*, recomenda-se o uso de colunas com alta cardinalidade como chaves de partição. Já as chaves de *clustering* são responsáveis por ordenar os dados dentro de uma partição, otimizando consultas que exigem filtros por faixas de valores (CASSANDRA, 2025) (CHEBOTKO e et., 2015). Por exemplo, em um sistema de registros de transações financeiras, a chave de partição poderia ser o *identificador do usuário*, enquanto a chave de *clustering* organizaria as transações por *data de ocorrência*, possibilitando consultas eficientes por período.

2.5. Agregados (Entidades e VOs)

O conceito de agregados é uma das ferramentas introduzidas pelo DDD que tem um papel fundamental na modelagem de dados com foco nas necessidades do negócio. Introduzidos por Eric Evans (2004), os agregados são definidos como uma unidade de consistência que agrupa um conjunto de objetos de dados relacionados, respeitando certas regras de negócio. Seu principal objetivo é simplificar e garantir a aplicação das regras dos invariantes de negócio. Dentro de um agregado, há um limite de consistência, o que significa que todas as operações realizadas dentro deste limite devem seguir as invariantes ou as regras específicas, independentemente das ações externas em cada estado do ciclo de vida. Quando uma transação é concluída, garante-se que tudo dentro do limite esteja em um estado consistente. Ou seja, um agregado agrupa dados relacionados que precisam ser tratados como uma unidade, tanto no armazenamento quanto no gerenciamento de

sua consistência. Um agregado é composto, geralmente, por entidades e objetos de valor (Value Objects ou VOs). As entidades são objetos que possuem uma identidade única, imutável ao longo do ciclo de vida da aplicação, mesmo que seus atributos mudem - por exemplo, um *cliente* com *CPF 123.456.891-10* continua sendo a mesma entidade mesmo que altere seu *nome* ou *endereço*. Já os VOs não possuem identidade própria e são definidos exclusivamente por seus atributos - como um *endereço* com *rua*, *número* e *cidade*; se dois endereços tiverem os mesmos dados, são considerados iguais. E para garantir a utilização correta dos agregados algumas práticas precisam ser seguidas (LIMA, 2016) (EVANS, 2004):

- Cada agregado possui uma entidade raiz que é responsável por verificar as invariantes e um limite que define o que precisa está dentro desse agregado;
- A raiz é o único objeto do agregado à qual os objetos externos têm permissão em fazer referência. Até a camada de persistência deve seguir essa regra, sendo necessário utilização de associações para busca ou alteração dos outros objetos do agregado.
- A entidade do agregado, que não é a raiz, só precisa ter identidade local, enquanto a entidade raiz precisa ter uma identidade global.
- Na operação de exclusão da raiz do agregado, todos os objetos desse agregado precisam ser excluídos.

Segundo Vaughn Vernon, a definição dos limites de um agregado exige cautela, pois envolve não somente as invariantes reais do modelo, mas também considerações sobre desempenho e escalabilidade do sistema. Muitas vezes, analistas de negócio especificam regras de negócio restritivas, sem avaliar os *trade-offs* na aplicação dessas regras. Agregados de grande grupos que geram registros volumosos, prejudicando a *performance* do BD, além de, em alguns casos, excederem limites de tamanho de armazenamento de registros do BD. Isso também pode comprometer transações em ambientes multiusuário, pois a probabilidade de muitos usuários tentarem modificar partes de um grande agregado ao mesmo tempo é maior. Por esses motivos, o agregado precisa ser o menor possível para garantir escalabilidade e desempenho da aplicação, mas também suficientemente grande para atender todas as invariantes reais (VERNON, 2016). Por exemplo, um sistema de *e-commerce* que informa de *pedidos* e *clientes* são armazenados no mesmo agregado. Nesse cenário, qualquer atualização em um *pedido* específico exigiria a atualização do

agregado inteiro, aumentando a concorrência entre transações e reduzindo o desempenho do sistema. Além disso, o crescimento dos *pedidos* ao longo do tempo tornaria o agregado grande, dificultando a recuperação eficiente dos dados. Para evitar esse problema, uma abordagem mais escalável seria separar *pedidos* e *clientes* em agregados distintos, permitindo que cada um evolua independentemente, reduzindo a contenção nas operações e melhorando a escalabilidade do BD.

Para manter o baixo acoplamento entre os agregados, é recomendável utilizar referências por identidade ao invés de referências diretas. Ou seja, ao invés de armazenar objetos inteiros de outros agregados, apenas os identificadores das entidades raízes desses agregados são armazenados como propriedades. Essa prática simplifica as operações transacionais, reduz o impacto no desempenho e está alinhada ao princípio de projetar agregados de forma independente (VERNON, 2016). Por exemplo, em um sistema de vendas, um *pagamento* pode referenciar um *pedido* apenas pelo seu *identificador*, em vez de armazenar todos os detalhes do *pedido* dentro do agregado de *pagamentos*. Isso reduz a complexidade das transações e melhora o desempenho e escalabilidade.

Para auxiliar na definição dos limites de um agregado, podem ser utilizados cálculos aproximados, conhecidos como *Back-Of-The-Envelope* (BOTE). Segundo Bentley, BOTEs podem ser utilizados para auxiliar a tomada de decisão no *design* de sistemas de TI. Segundo Vaughn Vernon, esses cálculos podem ser úteis para estimar de maneira simples e rápida o custo de um agregado, e avaliar se uma entidade deve ou não permanecer dentro de determinado agregado. Através dessa abordagem, mesmo não sendo um cálculo preciso, é possível antecipar problemas como aumento desnecessário de carga transacional ou sobrecarga em consultas e fornece um direcionamento para ajuste dos limites do agregado (VERNON, 2016) (BENTLEY, 1984). Por exemplo, ao projetar um agregado de *pedidos* em um *e-commerce*, pode-se calcular a quantidade média de pedidos que um cliente realiza por mês e, com base nisso, estimar o tamanho do agregado ao longo do tempo. Se o crescimento projetado indicar que o agregado se tornará muito grande e impactará a *performance* das consultas, isso pode sugerir a necessidade de dividir o agregado em estruturas menores.

Nas interfaces de usuário, é comum a necessidade de acessar informações de vários agregados para oferecer uma melhor experiência ao usuário. Esse cenário exige acesso a diversas tabelas, o que impacta negativamente a *performance* da aplicação. Caso essa sobrecarga se torne um

problema, pode valer a pena adotar o padrão CQRS (*Command Query Responsibility Segregation*) que separa comandos (responsáveis por modificar o estado do sistema) das consultas (responsáveis por recuperar dados). Essa abordagem permite otimizar as operações de leitura de forma independente, melhorando o desempenho e aumentando a flexibilidade do sistema. Por exemplo, pode ser utilizado *materialized views* ou até adotar outro BD para consulta com modelagem desnormalizada focada nas queries. No entanto, a aplicação de CQRS deve ser avaliada com cuidado, considerando a tolerância do sistema para atrasos na propagação das modificações (VERNON, 2016).

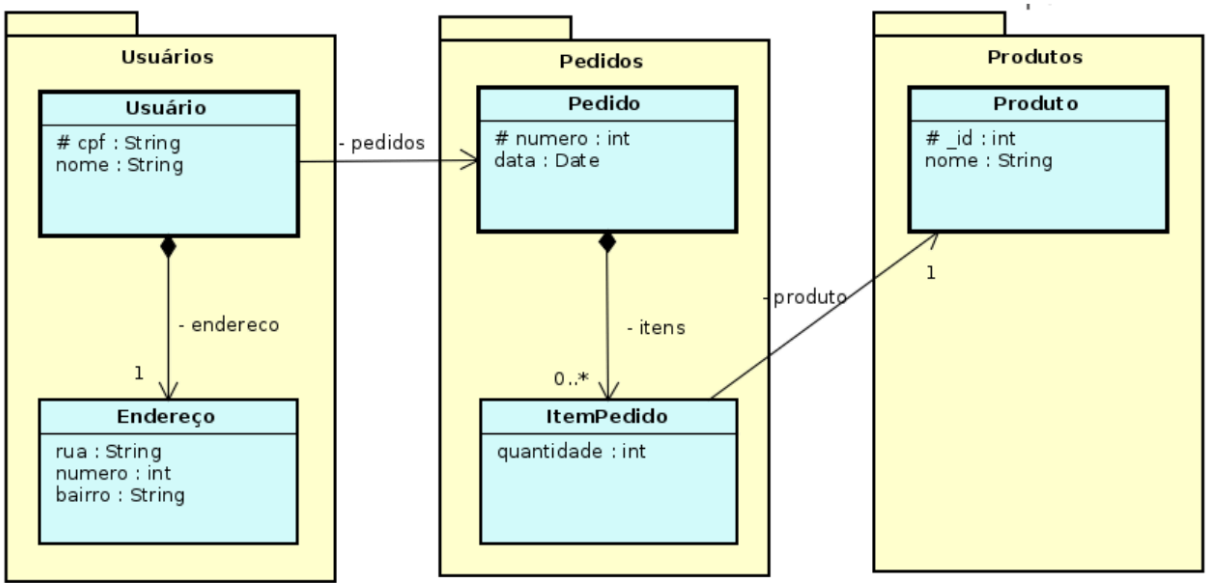
Essa abordagem de modelagem de agregados pode ser melhor implementada em BD *NoSQL*. Segundo Sadalage e Fowler, o conceito de agregados corresponde bem ao funcionamento da maior parte dos BDs *NoSQL*, devido à sua capacidade de armazenar estruturas agregadas de natureza flexível. Em comparação com os BDs relacionais tradicionais centralizados, os BDs *NoSQL* permitem que entidades e VOs relacionados sejam armazenados e manipulados juntos com a utilização de somente um nó por operação utilizando o conceito de modelagem de desnormalização dos dados. Além de conseguir aplicar o conceito de atomicidade e isolamento em todo agregado que garante maior consistência dos dados. Essa abordagem torna os *NoSQL* ideais para modelagem de agregados, especialmente em sistemas distribuídos, em que a escalabilidade e *performance* são cruciais (SADALAGE; FOWLER, 2013).

2.6. *Aggregate Modeling Language* (AML)

A *Aggregate Modeling Language* (AML) é uma linguagem de modelagem de dados voltada para BDs *NoSQL* que possuem estruturas de agregados. A AML é uma ferramenta para projetar e organizar dados na modelagem de entidades e VOs de acordo com o conceito de agregados do DDD. Por isso, a AML se apresenta como uma solução para modelagem de domínios (MONTEIRO, 2023). Como uma *Domain Specific Modeling Language* (DSML), a AML utiliza elementos visuais semelhantes ao *Unified Modeling Language* (UML), o que facilita sua adoção devido à popularidade do UML na modelagem de SIs. Além disso, a AML se beneficia de ferramentas de modelagem UML, tornando sua implementação mais acessível. Outra vantagem importante da AML é a baixa complexidade na sua utilização até comparando com o UML, pois conta com um número reduzido de construtores, e cada construtor tem uma semântica clara e única,

tornando a modelagem mais simples e direta (MONTEIRO, 2023). Por exemplo, a Figura 9 que apresenta uma modelagem lógica de domínio com 3 entidades (*usuário*, *pedido* e *produto*) e 2 VOs (*endereço* e *itemPedido*) representados em 3 agregados (*usuários*, *pedidos* e *produtos*).

Figura 9 - Exemplo de modelagem utilizando AML



Fonte - MONTEIRO (2023).

A AML é uma linguagem versátil, capaz de ser aplicada em diversos tipos de BDs *NoSQL* e cenários de uso. Seu conceito central, que é o conceito de agregados do DDD, é uma alternativa para modelar BD *NoSQL* com armazenamento flexível, como os orientados a documentos, família de colunas e chave-valor. Segundo Sadalage e Fowler, a aplicação desse conceito em BDs *NoSQL* oferece vantagens, como maior consistência e manter alto desempenho, assim atendendo às demandas de sistemas distribuídos e escaláveis (SADALAGE; FOWLER, 2013) (MONTEIRO, 2023).

Os construtores da AML são classificados em três categorias principais: pictogramas, nós e *links*. Esses elementos são fundamentais para modelar um BD *NoSQL* orientado a agregados para garantir uma estrutura clara e eficiente das entidades, VOs e seus relacionamentos (MONTEIRO, 2023).

2.6.1. Pictogramas

Os pictogramas na AML são símbolos gráficos que adicionam informações semânticas aos atributos e *links* sobre regras de unicidade, identidade e tipo de dados. Enquanto na UML, os pictogramas que estão associados a modificadores de visibilidade dos atributos de uma classe, na AML eles indicam diferentes restrições de unicidade e identidade aplicáveis a atributos ou relacionamentos

(MONTEIRO, 2023).

A Figura 10 apresenta uma tabela de pictogramas relacionados às regras de unicidade e identidade com seus respectivos nomes em inglês. A seguir, detalha-se cada pictograma apresentado na figura:

- **Pictograma de visibilidade pública (+) ou ausência de pictograma**: Indica um atributo ou relacionamento regular, sem restrições específicas de unicidade ou identificação.
    - Exemplo: A propriedade *nome* do tipo *string* da entidade *cliente.*
- **Pictograma de visibilidade protegida (#)**: Representa um identificador (do inglês *identifier)*, indicando que o campo é um identificador único da entidade. No Apache Cassandra, quando a chave primária contém uma única coluna, esse pictograma representa a chave primária e a de partição da tabela. Caso a chave primária tenha mais de uma coluna, esse pictograma representa apenas a chave de partição.
    - Exemplos:
        - A propriedade *ID*, do tipo *int*, que representa tanto a chave primária quanto a chave de partição da entidade clientes.
        - A propriedade *referência*, do tipo *string*, que representa a chave de partição na entidade *produtos*, pois sua chave primária é composta pelas colunas *referência* e *padrão*, nesta ordem.
- **Pictograma de visibilidade de pacote (~)**: Representa um discriminador (do inglês *discriminator*), usado para diferenciar registros entre si. Esse conceito é comum em entidades fracas e, no Apache Cassandra, corresponde à chave de ordenação (do inglês *Clustering Key*).
    - Exemplo: A propriedade *padrão*, do tipo *string,* que representa a chave de ordenação da entidade *produtos*, uma vez que sua chave primária é composta pelas colunas *referência* e *padrão*, nessa ordem.
- **Pictograma de visibilidade privada (-)**: Indica que um atributo ou relacionamento é único (do inglês *unique*), ou seja, não pode haver valores duplicados, impondo uma restrição de unicidade. Caso a propriedade for do tipo *array*, então a restrição de unicidade é aplicada

sobre os valores do *array* (equivalente a uma coleção do tipo *set* no Apache Cassandra). Se a propriedade tiver multiplicidade “1”, a restrição é aplicada diretamente na tabela, ou seja, só pode existir um objeto com o mesmo valor na tabela.

- Exemplos:
  - A propriedade *e-mail*, do tipo *string*, na entidade *cliente*, pois dois clientes não podem ter o mesmo e-mail.
  - A propriedade *telefones*, do tipo *set<string>,* na entidade *clientes* que se refere a um conjunto de telefones que não podem ser repetidos como *{ "+55 11 99999-0000", "+55 21 98888-1111" }*

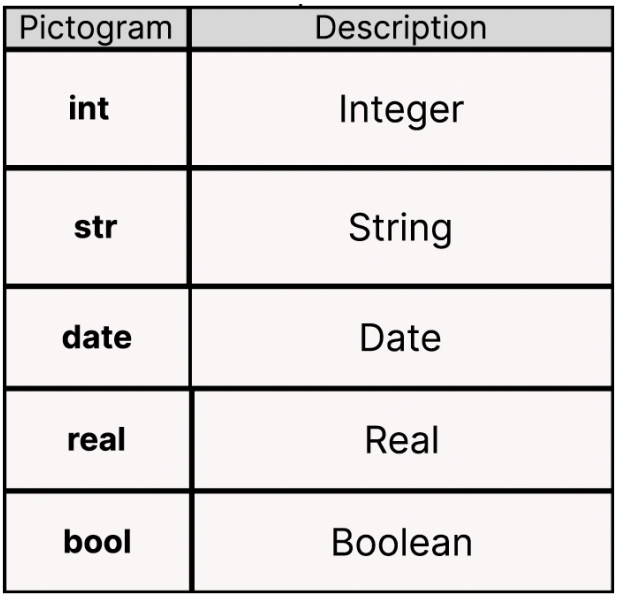
Figura 10 - Pictogramas de regras de unicidade e identidade da AML

Pictogram	Description
+	Regular
–	Unique
#	Identifier
~	Discriminator

Fonte - Arquivo do idealizador da AML.

A Figura 11 apresenta uma tabela com os pictogramas de tipos de dados e seus respectivos nomes. A AML abrange os principais tipos de dados comuns entre os BDs *NoSQL*.

Figura 11 - Pictogramas de tipos de dados da AML



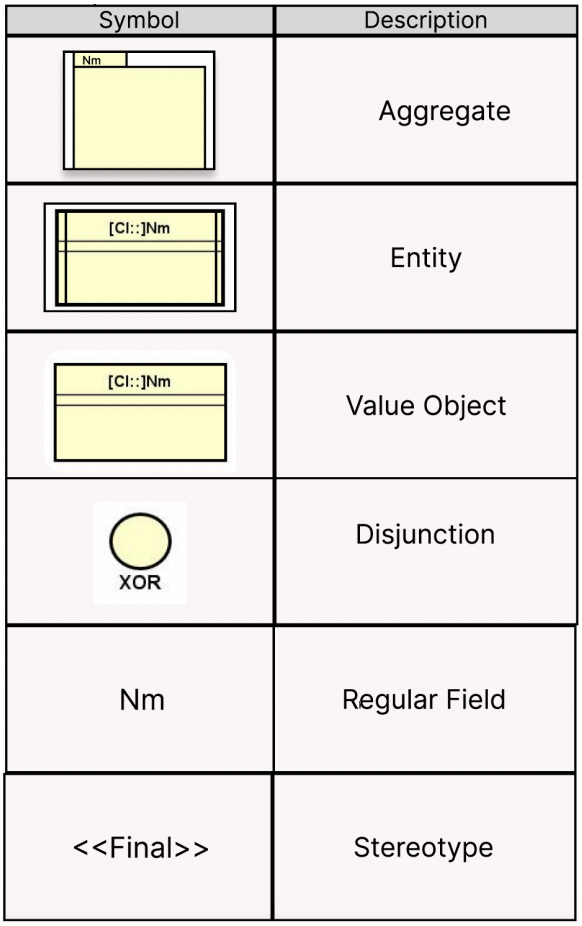
Pictogram	Description
int	Integer
str	String
date	Date
real	Real
bool	Boolean

Fonte - Arquivo do idealizador da AML.

2.6.2. Nós

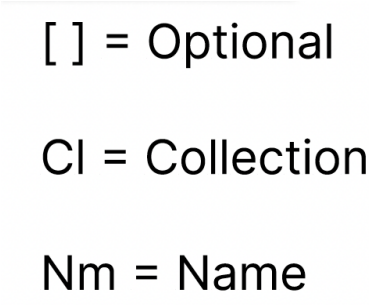
Os nós são elementos gráficos do UML responsáveis por representar os principais conceitos de modelagem, incluindo agregado (do inglês *aggregate*), entidade (do inglês *entity*), objeto de valor (do inglês *value object* - *VO*), disjunção (do inglês *disjunction*), atributo (*regular field*), e estereótipo (*stereotype*). Esses diagramas refletem a estrutura de dados de forma visual e intuitiva que são a base da modelagem em AML (MONTEIRO, 2023). A Figura 12 apresenta uma tabela com os elementos e seus respectivos nomes em inglês e a Figura 13 a legenda dessa tabela.

Figura 12 - Construtores do Nó na AML



Fonte - Arquivo do idealizador da AML.

Figura 13 - Legenda da figura dos Construtores do Nó na AML



Fonte - Arquivo do idealizador da AML.

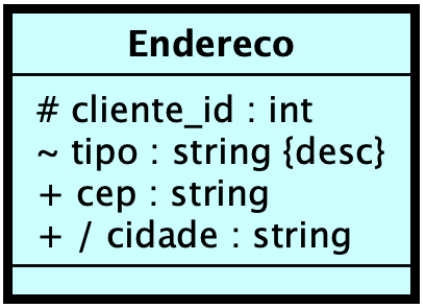
2.6.2.1. Atributo (Regular Field)

Na AML, um atributo é um campo associado a uma entidade ou um VO, representado graficamente

por meio de pictogramas que indicam sua regra de unicidade e identidade, seguido pelo nome do campo e pelo pictograma correspondente ao tipo de dado (MONTEIRO, 2023). A Figura 14 ilustra exemplos de atributos de uma entidade na AML. Além disso, a AML permite especificar características adicionais dos atributos, como multiplicidade, ordenamento e se o atributo é estático, por meio dos seguintes construtores:

- **Multiplicidade**: O símbolo "[*]" indica que o atributo é um array (multiplicidade N), enquanto sua ausência que indica que o atributo é simples (multiplicidade 1).
    - Exemplo: Conforme ilustrado na Figura 13, na entidade *endereço*, o atributo *complementos* representa uma lista de complementos associados ao endereço.
- **Ordenamento**: É especificado por meio do construtor constraints do UML "{}" após o tipo do atributo, como "{*desc*}" para ordenação decrescente, e "{*asc*}" ou ausência desse elemento para ordenação crescente. Esse ordenamento se aplica exclusivamente às colunas de discriminação, influenciando a forma como os registros são fisicamente organizados na BD. No Apache Cassandra, é o ordenamento definido nas *clustering keys* que definem a ordem dos registros dentro de uma partição.
    - Exemplo: Conforme ilustrado na Figura 13, na entidade *endereço*, o atributo *tipo de endereço* que é uma coluna de discriminação, está com símbolo "{*desc*}" que indica que os endereços do cliente serão ordenados de forma decrecente pelo seu tipo de "Z" até "A" na partição correspondente à chave primária.
- **Estático**: O atributo estático é indicado pelo símbolo "/" antes do nome do campo, utilizando a notação que na UML representa atributos derivados. No Apache Cassandra, esse símbolo representa uma coluna *STATIC* que o valor do atributo será compartilhado entre todas as linhas com a mesma chave de partição.
    - Exemplo: Conforme ilustrado na Figura 13, na entidade *endereço*, o atributo *cidade* está indicado que é uma coluna estática, ou seja, que a cidade é o mesmo em todas as linhas da partição de endereço do cliente.

Figura 14 - Exemplo de atributos na AML

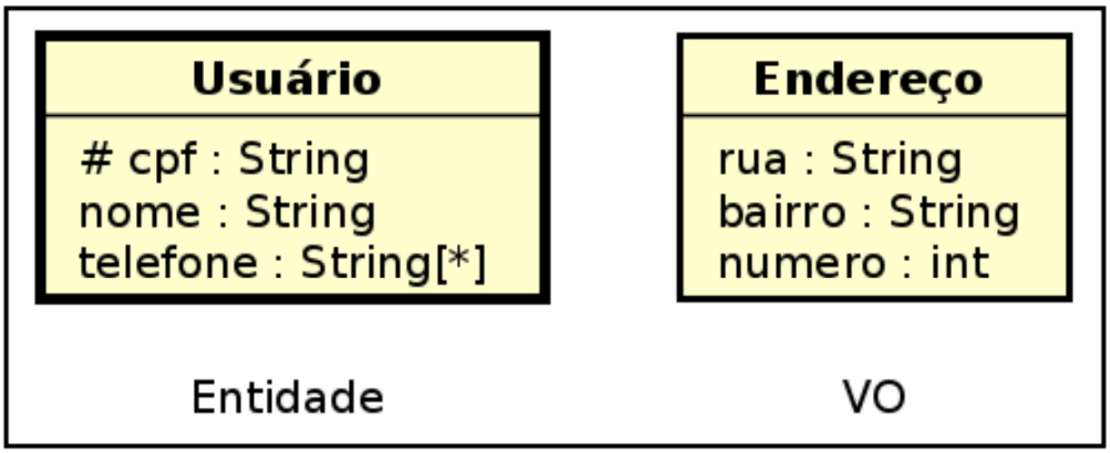


Fonte - Autoria própria.

2.6.2.2. Entidade (Entity) e Objeto de Valor (Value Object - VO)

Na AML, as entidades e os VOs são representados graficamente com base na anotação de classes do UML, diferenciando-se visualmente para facilitar a identificação de seus papéis no modelo. As entidades seguem o estilo da classe ativa do UML, sendo representadas por um contorno mais espesso. Por exemplo, a entidade usuário com identificador único CPF, segue essa representação, conforme ilustrado na Figura 15. Já os VOs são representados como classes comuns, com bordas mais finas, diferenciando-se das entidades. Um exemplo disso é o VO endereço conforme também demonstrado na Figura 15. Outra diferença visual dos construtores, é que uma entidade possui um identificador único, sendo representado graficamente por um pictograma de unicidade, enquanto um VO não possui identidade própria, sendo definido apenas pelos valores que armazena, (MONTEIRO, 2023).

Figura 15 - Exemplo de entidade e VO no AML



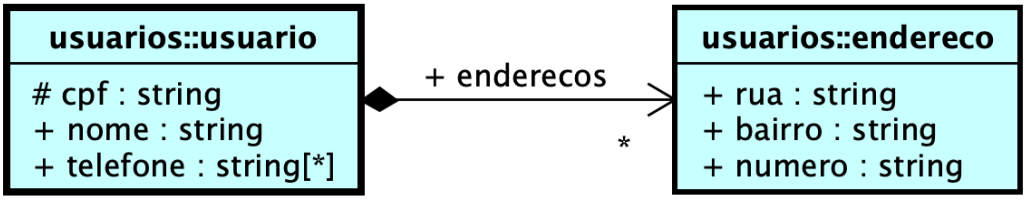
Fonte - MONTEIRO (2023).

2.6.2.3. Agregado (Aggregate)

Na AML, o agregado é o conceito central na modelagem de BD que consiste em um conjunto de entidades e VOs relacionados, com limites de consistência que asseguram o cumprimento das regras de negócio. Esses elementos são tratados como uma unidade lógica, o que significa que são armazenados em conjunto no BD. A AML oferece duas maneiras de representar graficamente um agregado (MONTEIRO, 2023):

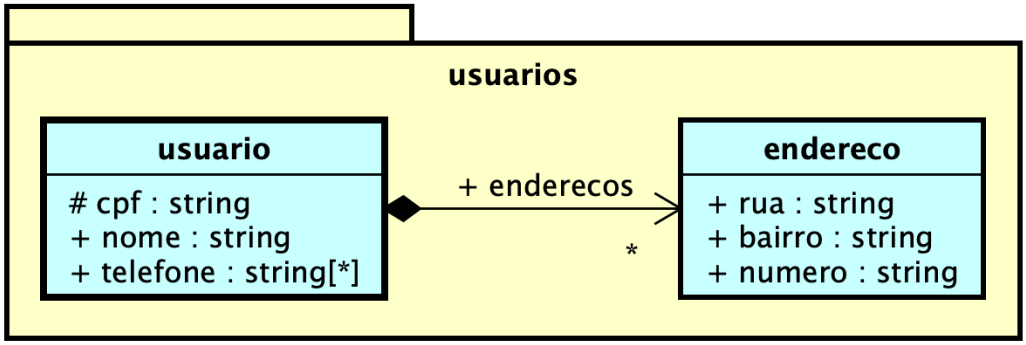
- **Prefixo no nome**: O nome do agregado é utilizado como prefixo dos nomes das entidades e VOs, seguido pelos caracteres "::", que, no UML, indicam o pacote ao qual a classe pertence. Por exemplo, no agregado *usuários*, a entidade raiz *usuário* e o VO *endereço* seguem essa nomenclatura, conforme ilustrado na Figura 16.
- **Símbolo do agregado**: A segunda forma, que será adotada neste trabalho, utiliza o símbolo de pacote do UML, onde o nome do agregado é explicitado diretamente no diagrama. Isso elimina a necessidade de incluir o nome do agregado nos nomes das entidades ou VO. A Figura 17 exemplifica essa abordagem com o agregado *usuários*, sua entidade raiz e o VO *endereço*.

Figura 16 - Exemplo agregado prefixo nome na AML



Fonte - MONTEIRO (2023).

Figura 17 - Exemplo agregado utilizando o símbolo na AML



Fonte - MONTEIRO (2023).

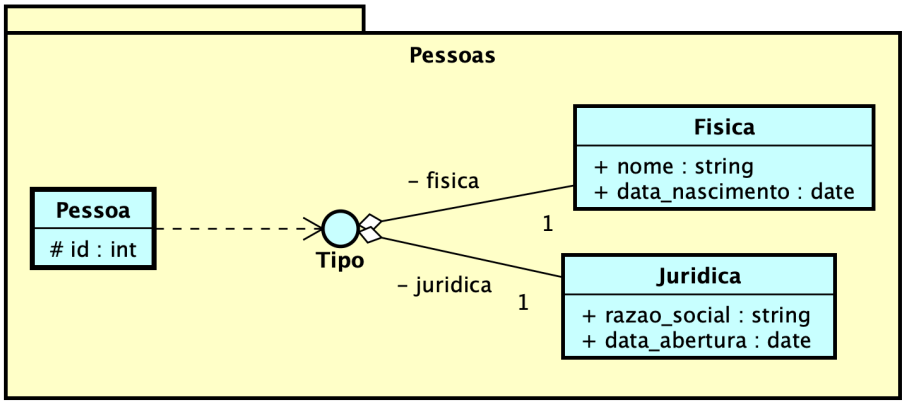
No Apache Cassandra, os dados do agregado são armazenados na mesma tabela. E para isso as entidades secundárias ou VOs que não sejam a entidade raiz precisam ser armazenadas em UDTs. Em alguns casos, pode ser necessário garantir a restrição de unicidade do identificador de uma entidade secundária dentro do agregado. Para isso, em vez de armazená-la como um *list* ou *set* de UDT, pode-se utilizar um *Map* de UDTs, onde a chave do mapa representa o identificador único da entidade. Por exemplo, se um agregado *usuários* possuir múltiplos *endereços*, e cada endereço precisar de um identificador único, a modelagem pode ser feita utilizando um *Map<String, UDT>*, onde a chave do mapa é o *identificador do endereço* e o valor é a estrutura da entidade armazenada como um UDT. Isso permite consultas mais eficientes e garante que não existam endereços duplicados para o mesmo usuário.

2.6.2.4. Disjunção (Disjunction)

Na AML, a disjunção (do inglês *disjunction*) é um elemento gráfico utilizado para representar relacionamentos mutuamente exclusivos entre entidades e VOs, sendo inspirado na notação de

interface do UML. Essa representação indica que uma entidade pode estar relacionada a apenas uma das opções disponíveis, mas nunca a ambas simultaneamente, sendo aplicada tanto para restrições de relacionamento quanto para generalização/especialização exclusiva. Visualmente, a entidade de origem estabelece um *link* até o símbolo de disjunção, que, por sua vez, se conecta às entidades ou VOs exclusivas, garantindo que, se uma entidade pode estar associada a A ou B, ela jamais poderá estar vinculada a A e B ao mesmo tempo. Por exemplo, no cadastro de clientes, onde um indivíduo pode ser registrado como *pessoa física* ou *pessoa jurídica*, mas nunca em ambas as categorias simultaneamente. A Figura 18 ilustra esse conceito, demonstrando como a AML representa graficamente essa relação de exclusividade.

Figura 18 - Exemplo do símbolo disjunção na AML



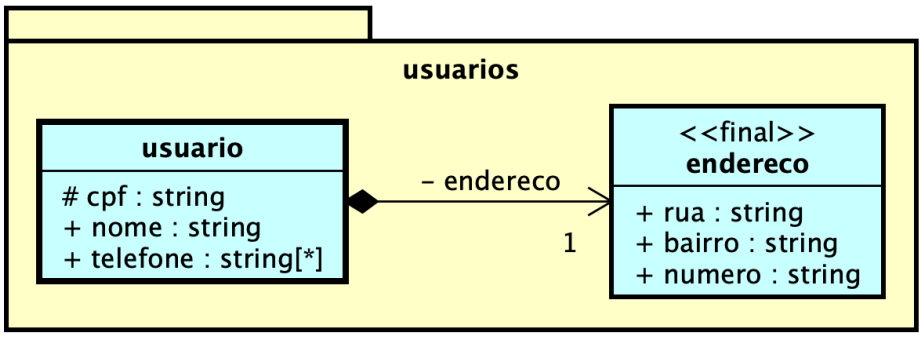
Fonte - Autoria própria.

2.6.2.5. Estereótipo (Stereotype)

Na AML, o estereótipo (do inglês *stereotype*) é um construtor baseado no símbolo de rótulo final do UML, utilizado para representar classes que não aceitam polimorfismo. Esse símbolo é representado entre *chevrons* (”«” e “»”) que é colocado em volta do nome “*final*”, indicando que ela é *final* e não pode ser herdada ou modificada. No contexto da AML aplicada ao Apache Cassandra, essa restrição se assemelha ao conceito de UDTs marcados como *frozen*, onde a estrutura do tipo fica imutável após a definição devido ao Cassandra tratá-los como um *blob*. Isso significa que, ao armazenar um UDT *frozen*, seus atributos não podem ser alterados individualmente, exigindo a reescrita completa da coluna sempre que uma modificação for necessária. Por exemplo, como a Figura 19 exemplifica essa representação com uma entidade de *cliente* que tem um UDT *endereço frozen* para garantir que todas as suas informações, como *rua*, *bairro* e *número*, sejam armazenadas

e recuperadas como um único bloco, evitando alterações parciais que possam comprometer a consistência dos dados.

Figura 19 - Exemplo do símbolo final na AML



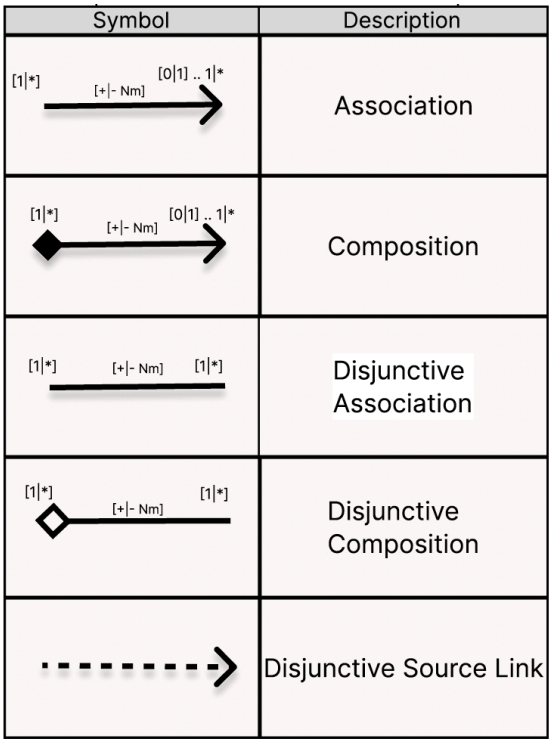
Fonte - Autoria própria.

2.6.3. Links

Os *links* são construtores da AML que representam os relacionamentos entre entidades e VOs, definindo como esses elementos se conectam dentro e fora de um agregado (MONTEIRO, 2023). Esses relacionamentos devem estar em conformidade com a regra do DDD de que, fora de um agregado, só é possível referenciar a entidade raiz do agregado. A AML fornece diversos tipos de *links*, cada um adequado a situações específicas, como associação, composição, disjunção e composição qualificada.

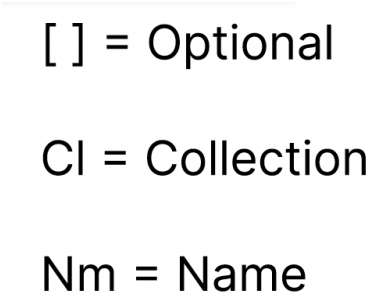
A Figura 20 apresenta parte desses construtores, com uma tabela contendo os símbolos e nomes originais em inglês com sua legenda na Figura 21. A Figuras 22 complementa esse conjunto, apresentando o outro tipo de *link* disponível na AML que é a composição qualificada, conforme definido por seu idealizador.

Figura 20 - Construtores de *link* de associação, composição e disjunção na AML



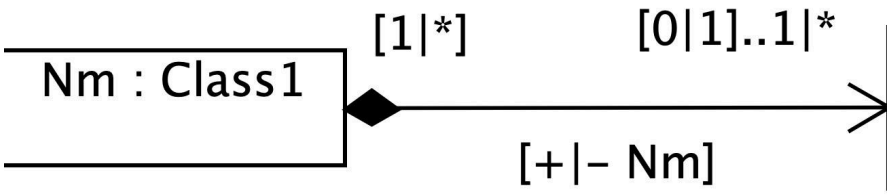
Fonte - Arquivo do idealizador da AML.

Figura 21 - Legenda da figura dos construtores de link de associação, composição e disjunção na AML



Fonte - Arquivo do idealizador da AML.

Figura 22 - *Link* de composição qualificada na AML

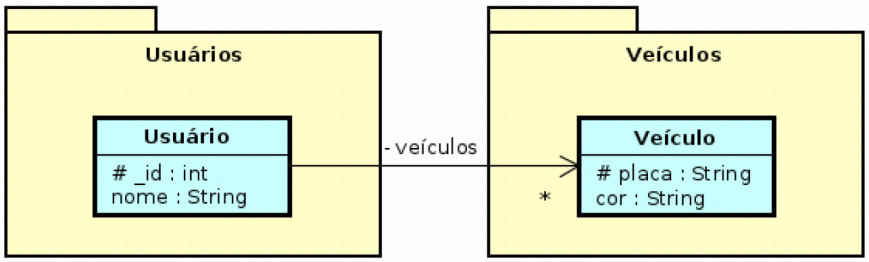


Fonte - Autoria própria.

2.6.3.1. Link de Associação (Association)

Representa relacionamentos entre entidades de agregados distintos, onde as entidades são referenciadas por identificadores, em vez de serem incorporadas diretamente. Ou seja, o relacionamento é por referência. No Apache Cassandra, não há suporte para relacionamentos diretos entre tabelas como ocorre em BDs relacionais. Além disso, não existem restrições de chave estrangeira para garantir a integridade referencial. Em vez disso, os identificadores são armazenados como atributos sem restrição, cabendo à aplicação garantir a consistência dos dados. Por exemplo, como demonstrado na Figura 23, o relacionamento de 1:N entre a entidade *usuário* e *veículo*, no qual o *veículo* armazena o *identificador do usuário*. A aplicação precisa garantir a consistência, como um registro cliente só pode ser excluído caso não tenha registro de veículo atrelado a ele.

Figura 23 - Exemplo de *link* de associação entre duas entidades



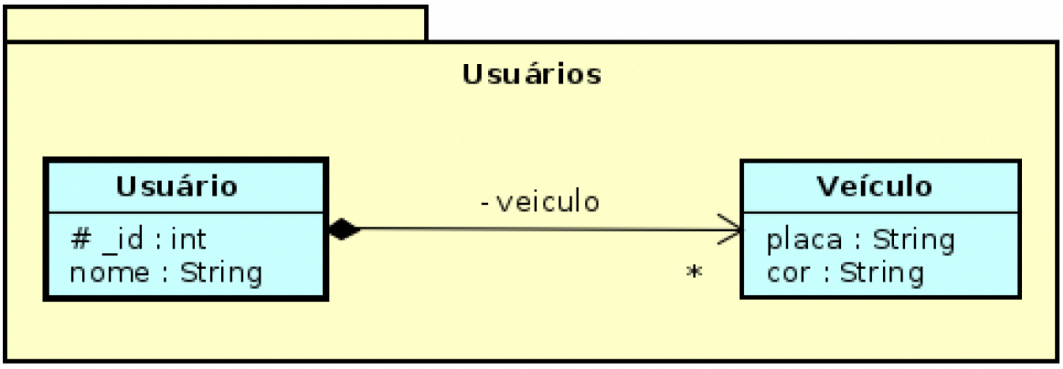
Fonte - MONTEIRO (2023).

2.6.3.2. Link de Composição (Composition)

Utilizado para representar relacionamentos entre entidades e VOs dentro de um agregado. O relacionamento de composição é por valor (ou cópia), portanto precisam ser armazenados juntos. No Apache Cassandra, a composição é realizada por meio da criação de um UDT para representar a

entidade ou VO que será composta e, para classe de origem, uma propriedade do tipo UDT para armazenamento da entidade ou VO. Por exemplo, como demonstrado na Figura 24, o relacionamento entre a entidade *usuário* e o VO *veículo*, nos quais a tabela *usuário* tem um atributo do tipo *list<Veículo>*, onde *veículo* é um UDT.

Figura 24 - Exemplo de *link* de composição entre uma entidade e um VO

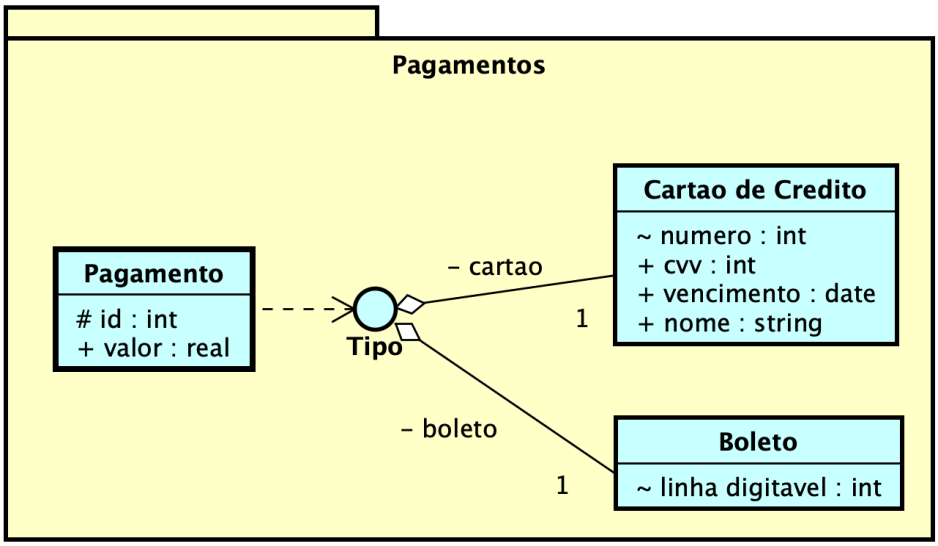


Fonte - MONTEIRO (2023).

2.6.3.3. Link de Origem Disjuntiva (Disjunctive Source Link)

Representa um relacionamento que define a entidade de origem de um vínculo mutuamente exclusivo entre entidades e VOs. Ou seja, a entidade de origem é a responsável por estabelecer restrições de relacionamento ou generalização/especialização exclusiva, garantindo que um elemento possa estar associado a apenas uma das opções disponíveis. Esse *link* conecta a entidade de origem ao símbolo de disjunção, que, por sua vez, se liga às entidades ou VOs que podem ser relacionadas por associação ou composição. Por exemplo, como demonstrado na Figura 25, um *pagamento* pode ser realizado via *cartão de crédito* ou via *boleto*, mas nunca por ambos ao mesmo tempo. A entidade *pagamento* é conectada ao símbolo de disjunção por meio do *link* de origem disjuntiva, e a disjunção se conecta às entidades *cartão de crédito* e *boleto*, garantindo que apenas uma dessas formas de pagamento possa ser registrada para cada transação.

Figura 25 - Exemplo de *link* de origem disjuntiva

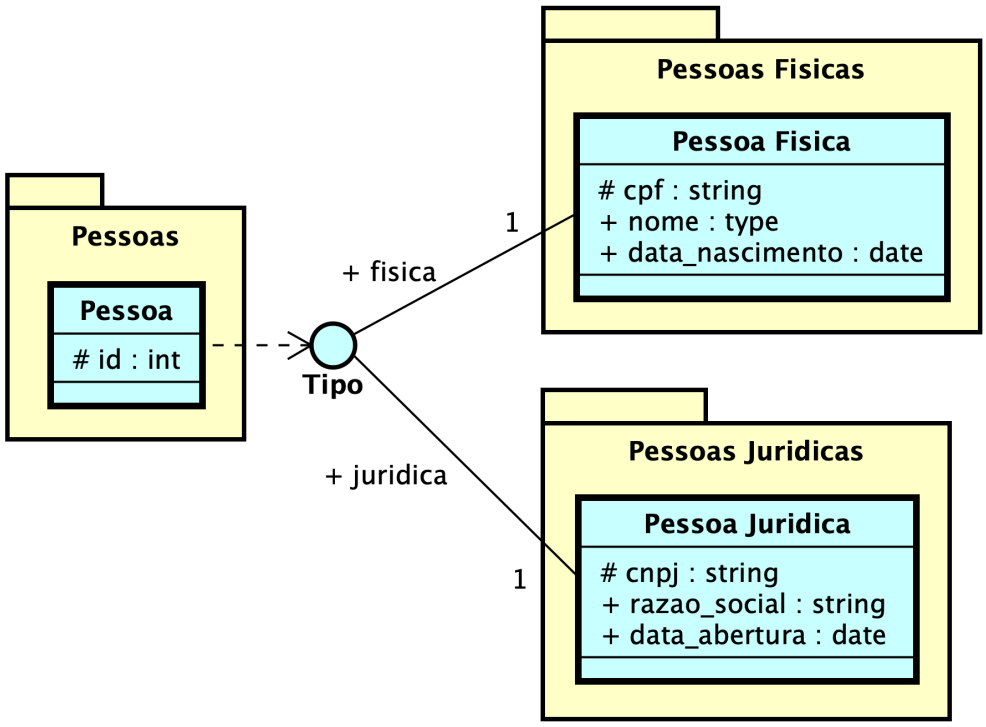


Fonte - Autoria própria.

2.6.3.4. Link de Associação Disjuntiva (Disjunctive Association)

A associação disjuntiva representa um relacionamento mutuamente exclusivo entre entidades por associação, em que uma entidade pode se relacionar com apenas uma das opções disponíveis, mas nunca com ambas simultaneamente. Esse tipo de associação é usado quando há restrições de relacionamento e generalização/especialização exclusiva, garantindo que uma entidade esteja vinculada a apenas um dos possíveis destinos dentro de um conjunto de opções. No Apache Cassandra, como não há suporte para restrições de chave estrangeira, a implementação desse conceito requer que o identificador da entidade relacionada seja armazenado como um atributo simples na tabela da entidade de origem. Para manter a exclusividade, a aplicação pode incluir uma *flag* de tipo ou armazenar os identificadores em colunas separadas, porém garantir que apenas um dos campos seja preenchido. Por exemplo, como demonstrado na Figura 26, uma *pessoa* pode estar associada a uma *pessoa física* ou a uma *pessoa jurídica*, mas nunca a ambas ao mesmo tempo. Existe um *link* de origem disjuntiva para conectar a entidade *pessoa* ao construtor disjuntivo para então se conectar as entidades *pessoa física* e *pessoa jurídica*. Neste exemplo, este *link* no Apache Cassandra pode ser representado com uma coluna de *tipo* na tabela *pessoa* para identificar se é pessoa física ou jurídica e uma coluna para armazenar o *ID* da *pessoa física* ou *jurídica*, ou criar duas colunas diferentes para armazenar o *ID* da *pessoa física* e *jurídica* para aplicar a restrição via aplicação.

Figura 26 - Exemplo de *link* de associação disjuntiva entre duas entidades

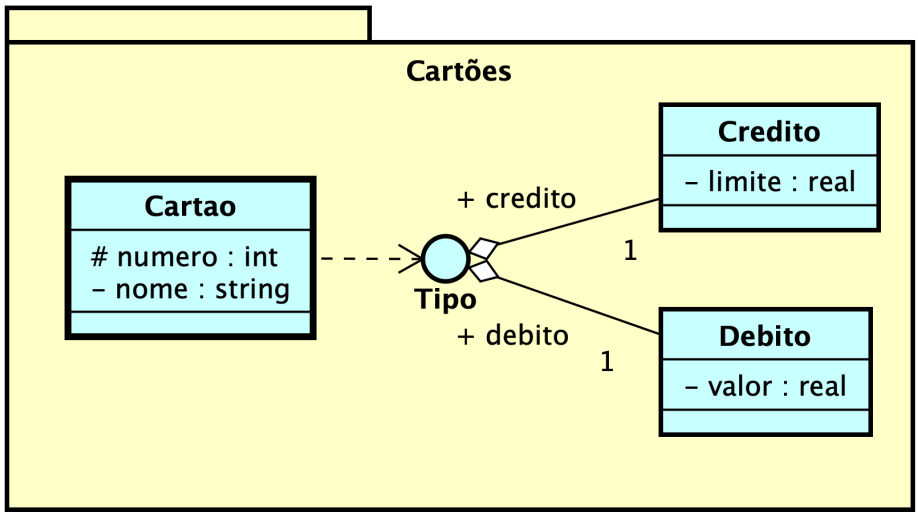


Fonte - Autoria própria

2.6.3.5. Link de Composição Disjuntiva (Disjunctive Composition)

A composição disjuntiva representa um relacionamento mutuamente exclusivo entre entidades ou VOs por composição, em que uma entidade pode possuir apenas um dos tipos possíveis de componentes, garantindo exclusividade dentro do agregado. E assim como o *link* de composição, deve ser tratado como uma unidade lógica, portanto precisam ser armazenados juntos. No Apache Cassandra, esse construtor pode ser representado com a criação de UDTs e atributos para armazenamento das entidades e VOs relacionadas, mas a garantia de exclusividade deve ser realizada pela aplicação. Por exemplo, como demonstrado na Figura 27, um *cartão* pode estar associado a um *cartão de crédito* ou um *cartão de débito*, mas nunca a ambos simultaneamente dentro do mesmo agregado. A entidade *cartão* se liga ao símbolo de disjunção, que então se conecta aos VOs *crédito* e *débito*, garantindo que apenas um desses componentes possa existir dentro do agregado em um determinado momento. Neste exemplo, este *link* no Apache Cassandra pode ser representado com armazenamento dos dois VOs em duas colunas distintas e ter uma coluna de *tipo* na tabela de *cartão* para identificar se é crédito ou débito e uma coluna.

Figura 27 - Exemplo de *link* de composição disjuntiva entre uma entidade e dois VOs

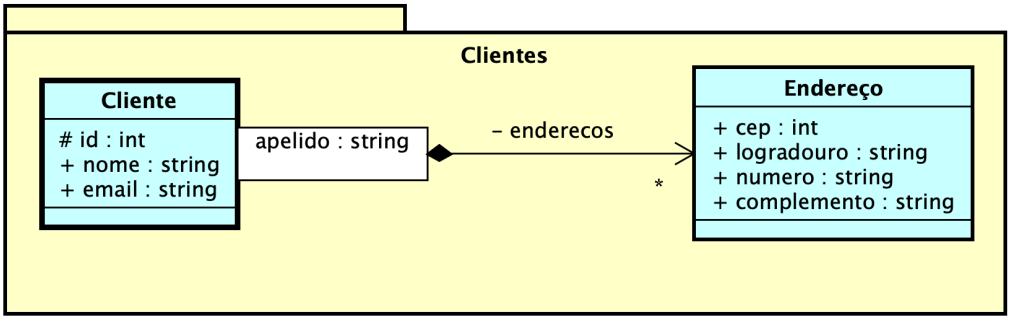


Fonte - Autoria própria.

2.6.3.6. Link de Composição Qualificada (Qualified Composition)

A composição qualificada representa um relacionamento entre entidades ou VOs que utiliza um qualificador - um ou mais atributos que servem para identificar unicamente os elementos do outro lado do relacionamento. Esse qualificador funciona como uma chave, permitindo acesso direto ao componente correspondente. Qualquer composição com multiplicidade N pode ser qualificada. Por se tratar de uma composição, os elementos envolvidos devem ser tratados como uma unidade lógica, portanto precisam ser armazenados juntos . No Apache Cassandra, esse tipo de modelagem é equivalente ao uso de um *Map* ou *Tuple*, onde a chave representa o qualificador e o valor representa o objeto composto, definido como um UDT. Por exemplo, como demonstrado na Figura 28, considere uma entidade *cliente* que pode possuir múltiplos *endereços*, cada um identificado por um *apelido* (ex.: residencial, comercial, entrega, etc.). Nesse caso, o relacionamento é representado como um *Map<String, Endereco>*, onde a chave do mapa é o *apelido* e o valor é a estrutura do VO *endereço* armazenada como um UDT. Essa modelagem garante que não existam endereços duplicados com o mesmo apelido para um cliente, e permite acessos diretos e eficientes a um endereço específico.

Figura 28 - Exemplo de *link* de composição qualificada entre uma entidade e um VO



Fonte - Autoria própria.

Esses *links* são essenciais para definir a hierarquia e os relacionamentos entre os elementos de agregados, garantindo que a modelagem lógica seja refletida na estrutura do BD. O AML oferece formas distintas de modelagem lógica para representar relacionamentos, que só é possível devido à flexibilidade do armazenamento de dados desnormalizado em BD *NoSQL*. A escolha do tipo de *link* deve ser feita para atender os requisitos da aplicação, considerando fatores como consistência, a forma e a frequência de acesso aos dados, a *performance* e as limitações do BD alvo. A seguir, serão apresentados cinco possibilidades de modelagem lógica para um relacionamento entre duas entidades ou uma entidade e um VO, e suas respectivas vantagens e desvantagens:

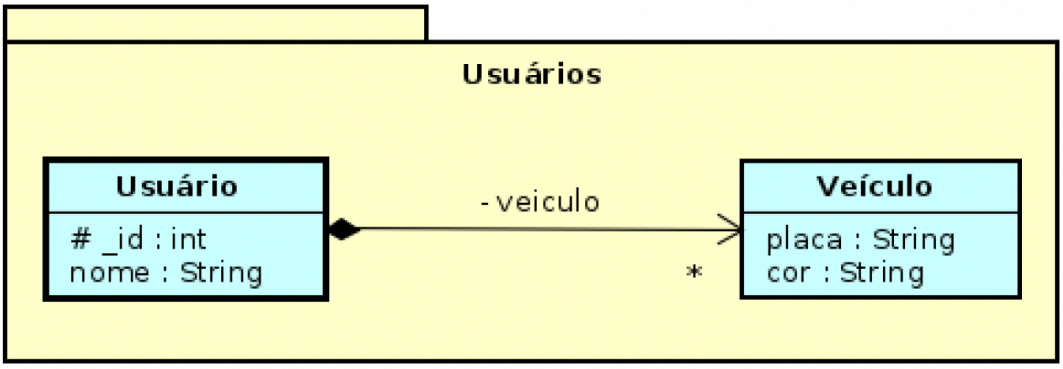
Agregado único

As entidades ou VOs que participam do relacionamento são armazenadas no mesmo agregado. Ou seja, essa forma garante que as entidades ou VOs sejam armazenadas na mesma instância ou nó do BD. Por exemplo, na Figura 29 que ilustra um relacionamento com cardinalidade 1-N entre entidade *usuário* e VO *veículo*.

- **Cenário ideal**: Relacionamento de entidades ou VOs diretamente relacionadas. Em operações de escrita e leitura, as entidades são acessadas juntas. A entidade precisa garantir regras invariantes com consistência transacional envolvendo o VO também.
- **Vantagem**: Respeitar característica do agregado (acesso junto), além de reduzir a necessidade de consultas adicionais.

- **Desvantagem**: Agregados grandes podem dificultar a escalabilidade do BD, além de atingir limitação de tamanho máximo (geralmente em MB) de cada registro no BD alvo.

Figura 29 - Relacionamento entre entidade e VO no mesmo agregado



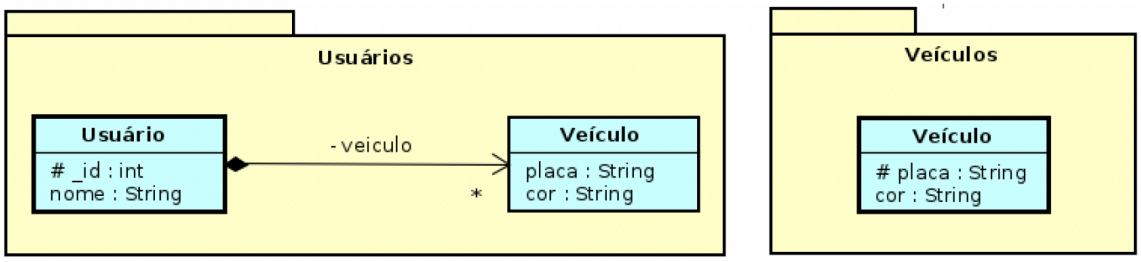
Fonte - MONTEIRO (2023).

Agregados com redundância da entidade

As entidades ou VOs que participam do relacionamento estão no mesmo agregado, porém uma delas (sem ser a raiz) está armazenada de forma redundante em outro agregado. Em caso de VO, a redundância será armazenada como entidade, pois a raiz do agregado precisa ser uma entidade para conseguir identificar o registro. Por exemplo, na Figura 30 que ilustra um relacionamento com cardinalidade 1-N entre a entidade *usuário* e o VO *veículo*, e uma redundância do VO *veículo* em outro agregado, porém representado como entidade.

- **Cenário ideal**: Uma das entidades ou VOs precisa ser acessada frequentemente e de forma isolada ou por diferentes formas de consulta. E a existência dessa entidade redundante é independente do relacionamento do primeiro agregado.
- **Vantagem**: Aumento da eficiência nas consultas. E acessar de forma rápida somente o necessário.
- **Desvantagem**: Aumento de complexidade para aplicação para garantir consistência entre as entidades e VOs duplicadas.

Figura 30 - Relacionamento entre entidade e VO no mesmo agregado com redundância do VO como entidade em outro agregado



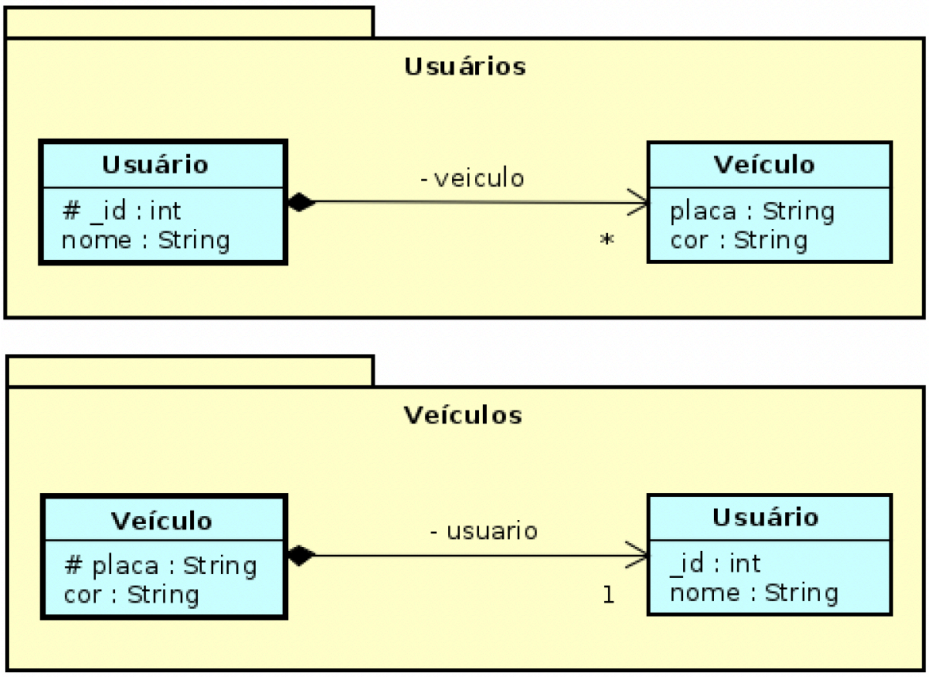
Fonte - MONTEIRO (2023).

Agregados complementares

As entidades ou VOs estão em um mesmo agregado e duplicadas em outro agregado, no qual a raiz do agregado é invertida e precisa ser uma entidade. Isso permite que as mesmas entidades ou VOs tenham padrões de acessos diferentes. Como ilustrado na Figura 31, no mesmo cenário que as abordagens anteriores de relacionamento de *usuário* e *veículos*, porém contando com uma redundância total do VO *veículo* em outro agregado.

- **Cenário ideal**: Quando as entidades ou VOs precisam ser acessadas juntas em diferentes formas de padrão de acesso.
- **Vantagem**: Maior flexibilidade nas consultas.
- **Desvantagem**: Aumento da complexidade para garantir consistência, além do aumento de armazenamento.

Figura 31 - Relacionamento entre entidade e VO no mesmo agregado com redundância em outro agregado com a raiz do agregado distinta



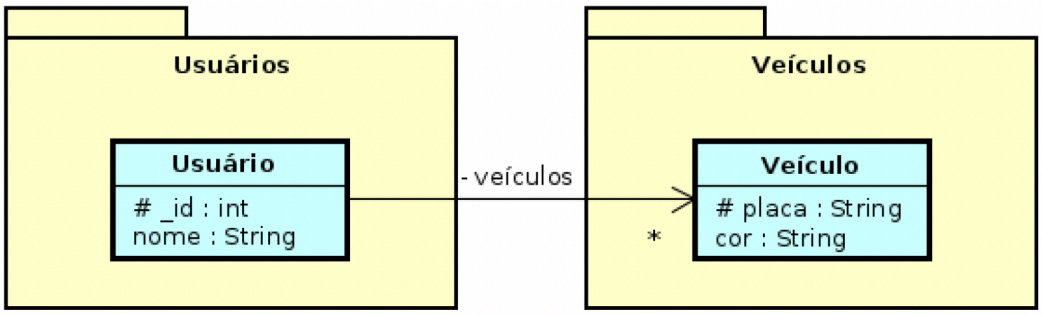
Fonte - MONTEIRO (2023).

Agregados referenciados

As entidades são armazenadas em agregados distintos e o relacionamento é mantido por referência. Por exemplo, na Figura 32, observa-se o relacionamento entre as entidades *usuário* e *veículo* em agregados próprios e distintos.

- **Cenário ideal**: Relacionamento sem acoplamento. Não seguem as regras de negócio e limites de consistência. Ou seja, são atualizadas e acessadas de forma independente.
- **Vantagem**: Maior escalabilidade, se de fato são acessados de forma independente.
- **Desvantagem**: Maior latência e necessidade de maior número de consultas, caso haja necessidade acesso aos dados das duas entidades.

Figura 32 - Relacionamento entre duas entidades em agregados distintos



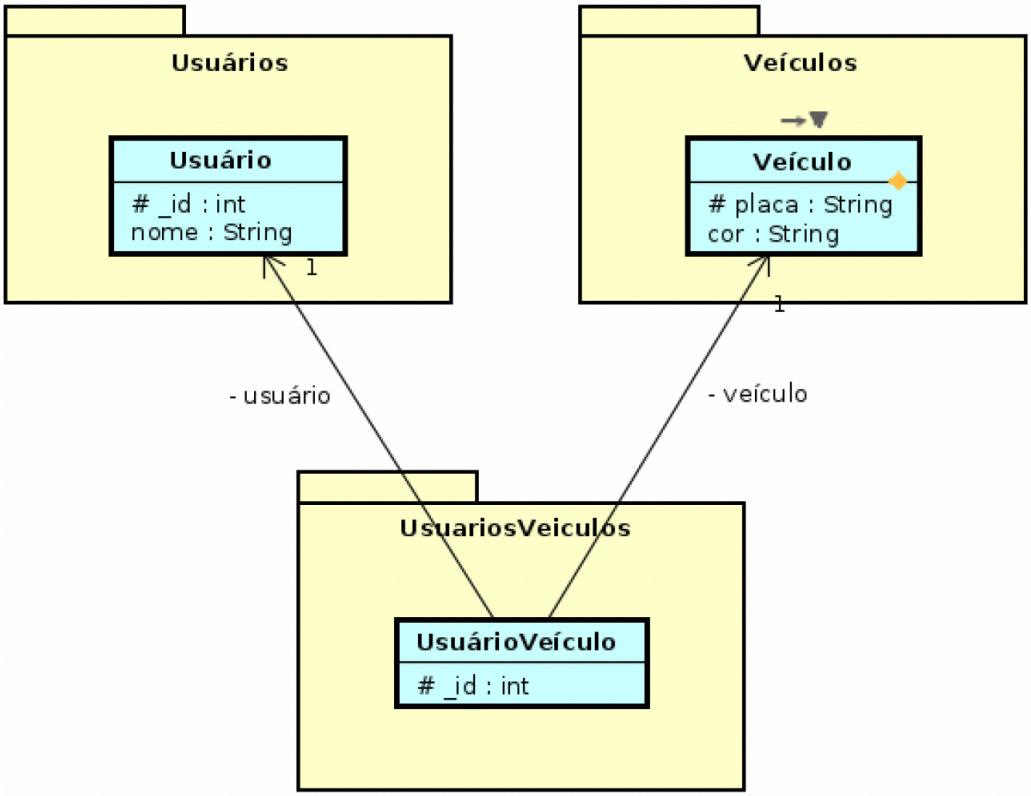
Fonte - MONTEIRO (2023).

Agregados intermediados

Criação de um agregado para armazenamento de informações de relacionamento por referência entre as duas entidades. Conforme ilustrado na Figura 33 que apresenta um agregado intermediário que possibilita até relacionamentos "N-N".

- **Cenário ideal**: Relacionamento n-n entre duas entidades.
- **Vantagem**: Permite relacionamentos mais complexos. Além de evitar duplicidade de dados.
- **Desvantagem**: Maior número de consultas pela necessidade de acesso às entidades e seus relacionamentos, caso haja necessidade acesso aos dados das duas entidades. Além de alguns BD *NoSQL* não tem suporte a utilização de *JOIN* como o Apache Cassandra.

Figura 33 - Relacionamento entre duas entidades com agregado intermediário



Fonte - MONTEIRO (2023).

Tabela 2 – Comparação das estratégias de armazenamento no AML

Estratégia de Armazenamento	**Cenário Ideal**	**Vantagens**	**Desvantagens**
Agregado único	Relacionamento diretamente relacionado; entidades acessadas juntas em operações de escrita e leitura.	Respeita as características do agregado. Reduz a necessidade de consultas adicionais	Dificuldade de escalabilidade com agregados grandes. Limitação de tamanho máximo do registro no BD.
Agregados com redundância da entidade	Uma entidade é acessada frequentemente de forma isolada ou com diferentes formas de consulta; existência independente.	Aumento da eficiência nas consultas. Acesso rápido ao necessário.	Maior complexidade para manter consistência entre as entidades duplicadas.
Agregados complementares	Entidades precisam ser acessadas juntas e de forma isolada em padrões de acesso diferentes.	Maior flexibilidade nas consultas.	Aumento da complexidade para manter consistência. Maior consumo de armazenamento.
Agregados referenciados	Relacionamento sem acoplamento; entidades acessadas e atualizadas de forma independente.	Maior escalabilidade	Maior latência. Necessidade de consultas adicionais para acessar ambas as entidades.
Agregados intermediados	Relacionamento n-n entre duas entidades.	Permite relacionamentos mais complexos.- Evita duplicidade de dados	Necessidade de múltiplas consultas para acessar entidades e seus relacionamentos. Alguns BDs *NoSQL*, como Apache Cassandra, não suportam *JOIN*.

Fonte - Autoria própria.

3. TRABALHOS RELACIONADOS

O aumento na demanda por BD *NoSQL* destaca a necessidade de definições claras e boas práticas de modelagem lógica para esses sistemas. Esse cenário tem incentivado contribuições acadêmicas para resolver tal problemática. Levando isso em consideração, e dado o objetivo deste trabalho de demonstrar de maneira prática a modelagem lógica de BDs *NoSQL* orientados a família de colunas utilizando a AML, destacam-se nesta seção três trabalhos com objetivos semelhantes desenvolvidos por Chebotko et al., Poffo e Monteiro.

O primeiro trabalho, de Chebotko et al., propõe uma metodologia de modelagem lógica voltada ao Apache Cassandra, levando em consideração as limitações e particularidades desse BD. O segundo trabalho, de Poffo, apresenta um esquema de conversão de um modelo conceitual para um modelo lógico destinado a BD orientados a família de colunas. Por fim, o terceiro trabalho, de Monteiro, explora a aplicação da AML na modelagem lógica de BD *NoSQL* orientados a documentos.

3.1. A Big Data Modeling Methodology for Apache Cassandra

O artigo "*A big data modeling methodology for Apache Cassandra*" de Chebotko e et. define uma metodologia de modelagem específica para o Apache Cassandra. Esse trabalho se destaca pela sua abordagem prática para enfrentar desafios de modelagem de dados em aplicações em ambientes de *Big Data*.

O objetivo principal do artigo é fornecer uma metodologia de modelagem de dados de fácil implementação que potencialize a eficiência do Apache Cassandra levando em consideração a sua arquitetura, forma de armazenamento e limitações do CQL. O trabalho auxilia para o desenvolvimento de um modelo lógico e físico para o Apache Cassandra que seja otimizado para operações de escrita e leitura com alta *performance* e escalabilidade horizontal eficiente.

A metodologia aplica um processo inverso ao tradicional processo de modelagem orientado a dados (*data-driven*). Essa abordagem é chamada de *function-driven*, pois a modelagem é orientada pelos requisitos da aplicação. Enquanto no processo *data-driven* a modelagem do BD é projetada com base em um modelo conceitual, como um diagrama entidade-relacionamento (DER), e posteriormente ajustada para atender aos requisitos de consulta, o método *function-driven* começa

analisando paralelamente o diagrama conceitual e as necessidades da aplicação. Dessa forma, a modelagem do BD é otimizada para as necessidades da aplicação, por exemplo para aplicação desnormalização quando necessário. Conforme ilustrado na Figura 34 que apresenta por meio de diagramas a diferença entre os processos de uma metodologia de modelagem de dados relacional e a metodologia proposta pelo trabalho.

A metodologia é baseada em Diagramas de Chebotko, ferramentas visuais que facilitam a modelagem de dados no Apache Cassandra. Nesse trabalho é aplicado uma modelagem de dados baseado nos requisitos da aplicação para o Apache Cassandra utilizando desses diagramas para a representação das estruturas de dados. Esse processo de modelagem é dividido em algumas etapas:

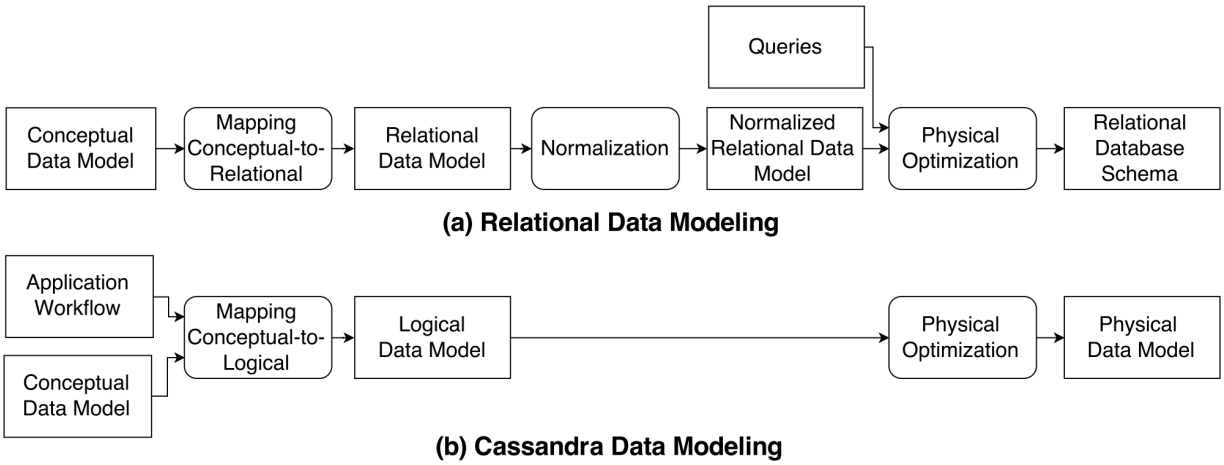
- **Identificação das entidades (*Conceptual Data Model*)**
    - Entender os dados e seus relacionamentos por meio de um modelo conceitual que é importante para a modelagem. Como exemplificado na Figura 35, um modelo conceitual de um domínio relacionado a *artigos digitais*, *reviews* e *usuários*.
- **Padrões de acesso (*Application Workflow*)**
    - Definir por meio de um diagrama, o fluxo das consultas e escritas que a aplicação realiza em cada tarefa. Por meio de setas que apontam para as *queries*/acessos que devem ser realizadas, com detalhamento de quais atributos serão utilizados, pesquisados, ordenados e agregados. Caso a aplicação seja uma aplicação *web*, pensar na ordem das chamadas de acordo com a ordem das telas. Como exemplificado na Figura 35, padrões de acesso de um domínio relacionado a *artigos digitais*, *reviews* e *usuários* que têm as *queries* de Q1 a Q9 com detalhamento de quais informações serão utilizadas e por quais propriedades serão realizadas os filtros e a ordenação.
- **Aninhamento de dados e desnormalização (*Mapping Conceptual to Logical*)**
    - Realizar a técnica de aninhamento de mais de uma entidade por meio de um critério, com objetivo de alcançar a forma de acesso de uma partição por consulta que é o acesso mais eficiente no Apache Cassandra. Cada tabela será representada por um quadrado com o nome da tabela e suas propriedades.
    - Usar desnormalização ou duplicação dos dados em várias tabelas para permitir a

aplicação realizar acessos de uma partição por consulta e melhorar a sua *performance*;

- Mapear as tabelas que suportem as consultas identificadas levando em consideração os critérios necessários: Pesquisa de igualdade, comparação e ordenamento;
- Definir as chaves primárias que incluem a de partição, por meio do construtor "K", e do *clustering*, por meio do construtor "C↑" e "C↓", para otimizar a distribuição dos dados e o desempenho das consultas.
- Por exemplo, a Figura 36 apresenta a modelagem lógica utilizando o Diagrama de Chebotko de um domínio relacionado a artigos digitais, *reviews* e usuários. Os quadrados representam as tabelas que se baseiam nas informações das *queries* detalhadas no padrão de acesso da aplicação. E as setas representam a ordem das chamadas da aplicação, neste exemplo, a aplicação inicia-se pela chamada da Q1 ou Q2. Essas chamadas utilizam as tabelas *Artifacts_by_venue* ou *Artifacts_by_author*. Após uma dessas chamadas, a aplicação pode chamar a Q3, Q4 ou Q5 que cada uma consulta as tabelas *Users_by_artifact*, *Experts_by_artifact* ou *Ratings_by_artifact* respectivamente. Com isso, o cliente poderá consultar as Q6, Q7 ou Q8 que acessam as tabelas *Venues_by_user*, *Artifacts_by_user*, *Reviews_by_user* respectivamente. E por fim, a aplicação realiza a consulta Q9 que utiliza a tabela *Artifacts*, e posteriormente retornar ao fluxo pelas chamadas Q3, Q4 ou Q5, e assim sucessivamente.

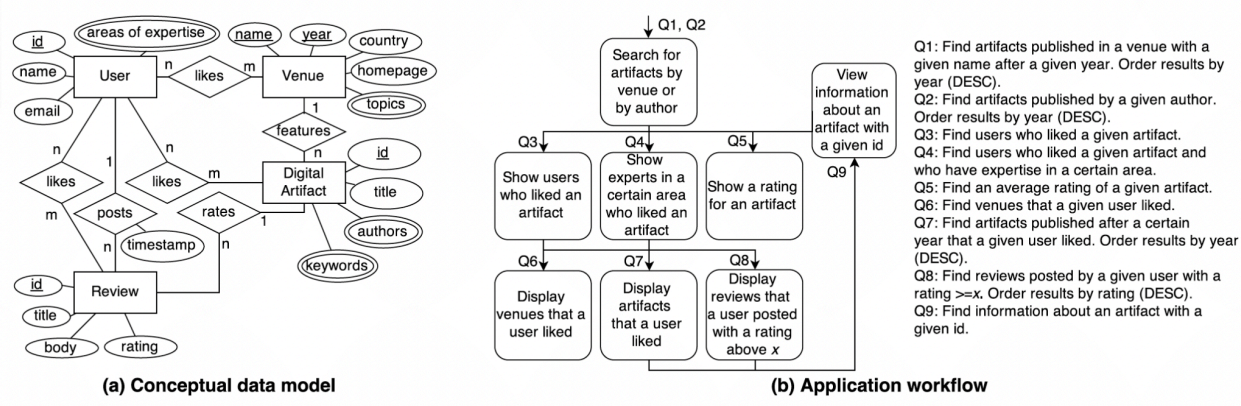
- **Mapeamento de entidades para tabelas (*Physical Optimization*)**
  - Mapear o modelo lógico para modelagem física no Apache Cassandra;
  - Otimizar tabelas, principalmente remodelando chaves primárias, *clustering* e *indexes* para melhorar a *performance* das consultas do Apache Cassandra.
  - Validar esquema proposto por meio de casos de uso reais e testes de desempenho;
  - E conforme a necessidade, ajustar o esquema com base no resultado dos testes.

Figura 34 - Comparação entre modelagem de dados relacional e a metodologia proposta por Chebotko (2015)



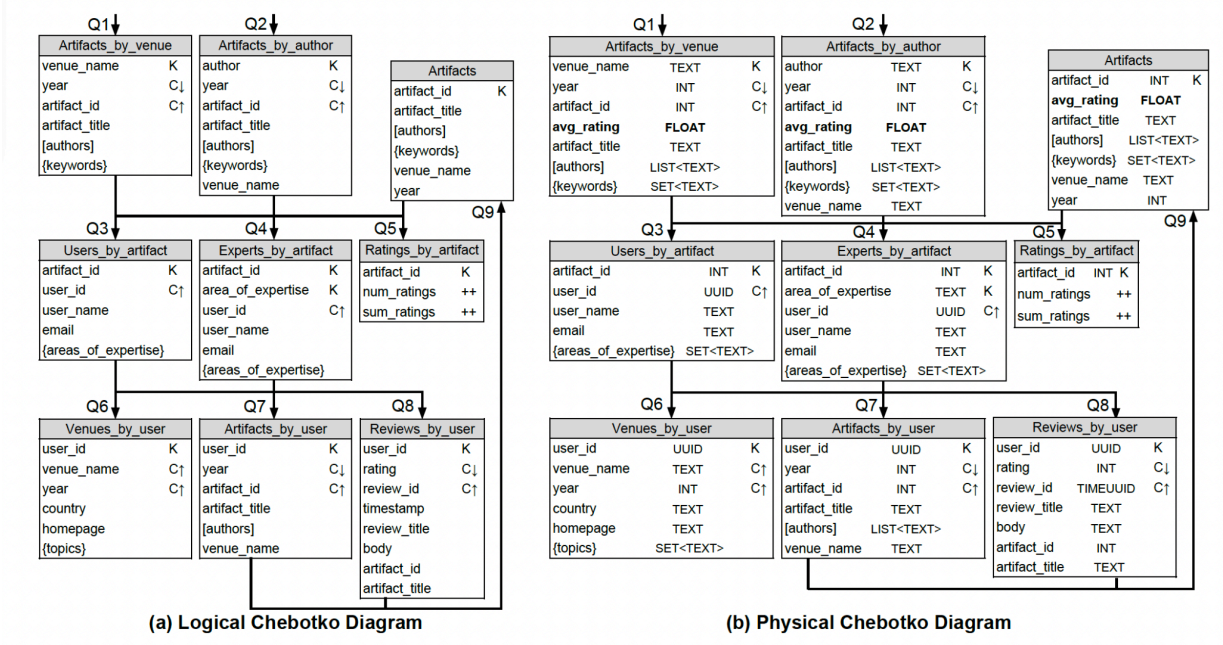
Fonte - CHEBOTKO e et. (2015).

Figura 35 - Exemplo de esquema conceitual e fluxo da aplicação (2015)



Fonte - CHEBOTKO e et. (2015).

Figura 36 - Exemplo de modelagem lógica e física utilizando Diagrama de Chebotko (2015)



Fonte - CHEBOTKO e et. (2015).

Segundo Chebotko et al., esse trabalho fornece uma modelagem que inova pela prioridade dada aos padrões de acesso e a utilização de uma estrutura desnormalizada de dados. E destaca a técnica de visualização para captura de modelos lógicos e físicos, chamada Diagramas de Chebotko.

Uma lacuna desse trabalho de Chebotko é a sua especificidade para o Apache Cassandra. A metodologia proposta é adaptada às características e limitações do Apache Cassandra, tornando sua implementação difícil, não apenas em outros BDs *NoSQL*, mas também em qualquer outro BD *NoSQL* orientado à família de colunas como Apache HBase ou ScyllaDB.

3.2. Projeto Lógico de BD *NoSQL* Colunares a partir de Esquemas Conceituais Entidade-Relacionamento Estendido (EER)

O trabalho "Projeto lógico de BD *NoSQL* colunares a partir de esquemas conceituais entidade-relacionamento estendido (EER)" de João Paulo Poffo apresenta uma conversão de modelos conceituais de dados para modelos lógicos para BDs *NoSQL* orientados a colunas. Esse trabalho preenche a lacuna existente entre as etapas de modelagem conceitual e lógica do projeto de BD para *NoSQL* colunar.

O principal objetivo do artigo é desenvolver um processo de conversão de modelos conceituais baseados em EER, para esquemas lógicos específicos para BDs *NoSQL* orientado a colunas. Este trabalho visa facilitar a adoção de BDs *NoSQL* colunares como Apache Cassandra e Apache HBase.

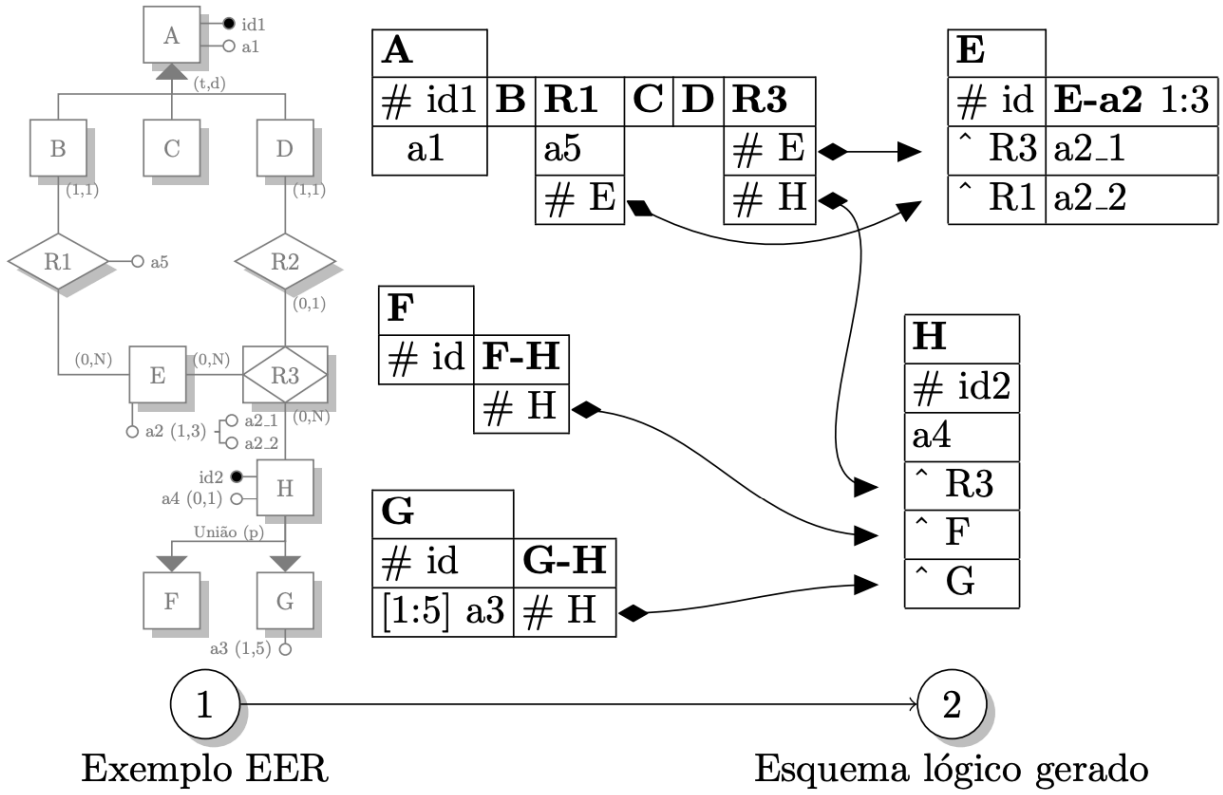
O processo de conversão proposto por Poffo é detalhado em etapas sistemáticas que preservam a integridade dos dados e a eficiência para consultas aos dados no BD *NoSQL* colunar. Esse processo é muito semelhante ao processo de conversão de Lima (2016) que também utiliza o esquema conceitual EER e as informações de carga de dados para conversão. Suas principais etapas são:

- **Definição do modelo conceitual:**
  - Definir o modelo conceitual de EER para representar entidades, relacionamentos e atributos;
  - Identificar as entidades, as chaves e os relacionamentos importantes para a aplicação.
  - Por exemplo, na Figura 37 que apresenta um exemplo de modelo EER com algumas entidades e relacionamentos.
- **Definição da modelagem de carga:**
  - Definir o modelo de carga das entidades e relacionamento de acordo com seu volume de dados e carga da aplicação.
- **Conversão das entidades para tabelas:**
  - Converter as entidades EER em tabelas do BD;
  - Utilizar a modelagem de carga para mapear e dar apoio ao processo de modelagem para garantir a *performance*;
  - Definir as chaves primárias e de *clustering* para melhorar a *performance* das consultas aos dados.
  - Mapear os relacionamentos 1-N e N-N para estruturas de tabelas apropriadas;
  - Desnormalizar para melhorar a *performance* das consultas, mesmo que isso resulte em duplicação de dados;

- Normalizar os dados para tratar redundâncias desnecessárias que não comprometam a *performance* das consultas.
- Por exemplo, na Figura 37 apresenta no modelo lógico as tabelas *A, E, F, G* e *H*, que foram definidas por meio do processo de conversão desse trabalho que utiliza como apoio a modelagem de carga para definir quais entidades serão transformadas em tabelas de fato ou desnormalizadas. Além disso, foi definido as chaves primárias e estrangeiras por meio do operador "#". E os relacionamentos por meio de figuras de relacionamento de composição da UML.

- **Validação da modelagem lógica:**
  - Validar o modelo lógico através de análise da consistência e desempenho;
  - Caso necessário, realizar ajustes baseados nos resultados dos testes para garantir maior eficiência e escalabilidade.

Figura 37 - Exemplo de esquema lógico gerado pelo processo de conversão proposto por Poffo e et.



Fonte - POFFO (2016).

Segundo Poffo, essa metodologia de conversão é uma metodologia promissora que busca criar um modelo lógico por meio do modelo conceitual e modelagem de carga que apresenta resultados promissores e melhores referente a *performance* das consultas que foram experimentadas comparados a modelagem de agregados.

Apesar das contribuições significativas, uma lacuna notável do trabalho de Poffo é sua abordagem específica para BDs *NoSQL* orientados a colunas. Idealmente, uma linguagem de modelagem comum aplicável a diferentes tipos de BDs *NoSQL* (colunares, documentais, chave-valor, etc.) proporciona maior flexibilidade, uniformidade e padronização na modelagem de dados.

3.3. Modelagem de Banco de Dados Orientados a Documentos com AML

O trabalho "Modelagem de banco de dados orientados a documentos com AML" de Thomas Anderson Feitosa Monteiro explora a aplicação da AML na modelagem BD *NoSQL* orientados a documentos. Este estudo é relevante por investigar a utilização do AML - uma linguagem fundamentada em conceitos de DDD - para melhorar o processo de modelagem lógica de BDs *NoSQL* documentais.

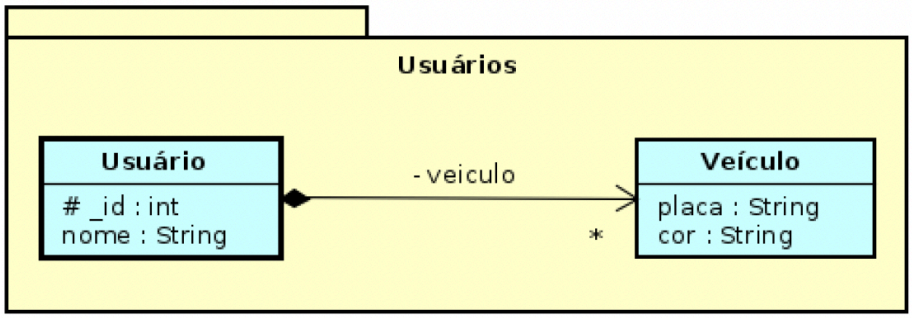
O trabalho busca a implementação dessa linguagem com objetivo de encontrar possíveis limitações da utilização na modelagem lógica em BDs *NoSQL* documentais. Além de apresentar a AML e seus construtores com conceitos de DDD comuns e essenciais na implementação na modelagem de armazenamento de dados. Monteiro com um estudo experimental e prático aplicando AML em diversos cenários no MongoDB. O experimento apresenta algumas etapas que são:

- **Definição de requisitos de negócio:**
  - Identificar e definir dos requisitos de negócio e das operações de dados;
  - Realizar mapeamento dos agregados do DDD no modelo de dados.
- **Modelo conceitual:**
  - Definir as entidades e os VOs entre agregados;
  - Definir os relacionamentos entre as entidades e os VOs.
- **Mapeamento para modelo lógico (Aplicação da AML):**
  - Converter os modelos conceituais em esquemas lógicos utilizando AML;
  - Estruturar em documentos de maneira otimizada para consultas e operações de escritas, mantendo a integridade e consistência dos dados.
  - Por exemplo, na Figura 38 que apresenta um agregado de *usuário* com entidade *usuário*, um VO *veículo* com relacionamento por composição, suas propriedades e identificadores. Esse diagrama representa estruturas que serão armazenadas em uma mesma coleção no MongoDB para manter a unicidade lógica do agregado. Com isso, mantém uma consistência entre os dados de *veículo* e *usuário* por serem armazenados

juntos.

- **Implementação e testes:**
  - Implementar modelos lógicos em um BD *NoSQL* orientados a documentos (como MongoDB).

Figura 38 - Exemplo de modelo lógico utilizando AML por Monteiro.



Fonte - Monteiro (2023).

O trabalho de Monteiro oferece várias contribuições para a área de modelagem de dados em BD orientado a documentos, oferecendo uma abordagem prática e estruturada. Contribuições essas incluindo:

- **Aplicação prática da AML:**
  - Demonstra como a AML pode ser utilizada na prática para modelar BDs documentais, mostrando sua flexibilidade e aplicabilidade.
- **Integração de DDD com *NoSQL*:**
  - Implementação de conceitos fundamentais de DDD na modelagem *NoSQL* proporcionando uma estrutura de dados mais estruturada, consistente e orientada ao domínio de negócio.

O trabalho apresenta uma aplicação prática de AML especificamente em BDs *NoSQL* documentais, porém há a necessidade de aplicação dessa linguagem em outros tipos de BDs *NoSQL* para validar sua aplicabilidade, principalmente em BDs *NoSQL* orientado a família de colunas devido a sua popularidade.

3.4. Comparação entre o trabalho proposto e os relacionados

Tabela 3 – Comparação entre o trabalho proposto e os relacionados.

Critério	**CHEBOTKO, Artem; KASHLEV, Andrey; LU, Shiyong [7]**	**POFFO, João Paulo et al. [8]**	**MONTEIRO, Thomas Anderson Feitosa [9]**	**Trabalho Proposto**
Ano	2015	2016	2023	2025
Banco de dados	Apache Cassandra	Colunares	MongoDB	Apache Cassandra
Linguagem de modelagem	Diagrama de Chebotko	Modelo lógico documento	AML	AML
Escopo	Definição de uma modelagem lógica, por meio de diagramas, específica para o Apache Cassandra.	Definição de um processo de conversão de um modelo conceitual para uma modelagem lógica correspondente para BDs colunares.	Avaliar a aplicação do AML na modelagem lógica de BD orientado a documentos, uma linguagem baseada nos conceitos de DDD, com objetivo de abranger diversos tipos de bancos *NoSQL*	Avaliar a aplicação do AML na modelagem lógica em BD *NoSQL* orientados a família de colunas

Fonte - Autoria própria.

Enquanto esses trabalhos fornecem contribuições para a modelagem lógica de BD *NoSQL*, o trabalho proposto detalha a aplicação da AML em um BDs *NoSQL* orientados a família de colunas (Apache Cassandra) devido ao potencial da AML para modelagem lógica em diversos tipos de BDs

NoSQL. Isso porque a linguagem fundamenta-se em princípios comuns e essenciais que são importantes para todos os tipos de BDs *NoSQL*. Isso permitiria que a mesma metodologia ou linguagem de modelagem lógica fosse utilizada em diversos contextos e BD.

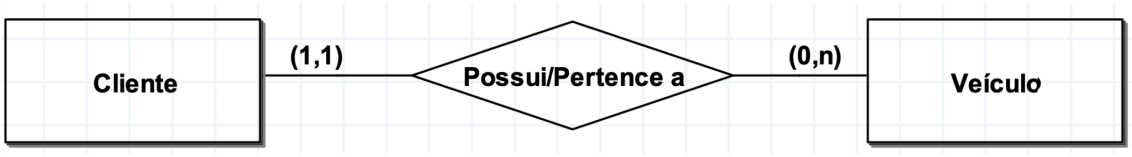
4. METODOLOGIA

4.1. Explicação da Metodologia do Experimento

A metodologia adotada neste trabalho consiste na aplicação prática da modelagem lógica utilizando a AML no BD Apache Cassandra. O objetivo é avaliar se os construtores da AML são adequados para expressar as características e restrições do Cassandra, como atomicidade por linha, necessidade de desnormalização, estrutura de partição e uso de tipos nativos e personalizados (ex.: UDTs, *collections*, etc.). Além disso, busca-se investigar se o Cassandra é capaz de implementar os construtores da AML de forma funcional e coerente com sua arquitetura distribuída.

A capacidade de armazenar estruturas complexas é uma das principais características dos BDs *NoSQL*, permitindo que uma mesma modelagem conceitual seja traduzida em diferentes estruturas no modelo lógico. Essa flexibilidade está diretamente alinhada ao conceito de agregados do DDD. Com base nesse princípio, o experimento implementa oito abordagens distintas de modelagem lógica utilizando a linguagem AML no Apache Cassandra, todas fundamentadas em um mesmo modelo conceitual que representa um relacionamento de cardinalidade "1:N" entre as entidades *cliente* e *veículo*. Como ilustrado na Figura 39, o cenário modelado envolve um *cliente* que pode possuir nenhum ou vários *veículos*, enquanto cada *veículo* pertence a um único *cliente*. A partir desse modelo conceitual, diferentes estruturas lógicas foram derivadas, variando aspectos como desnormalização de dados, uso de referências, aplicação de redundância e organização das consultas, com o objetivo de avaliar o mapeamento prático dos construtores da AML no Apache Cassandra. Por isso, as abordagens foram construídas explorando os principais construtores da AML - como *links* de composição, composição qualificada, associação, atributos com multiplicidade, referências e elementos redundantes.

Figura 39 - Modelagem conceitual do experimento.



Fonte - Autoria própria.

A escolha da estrutura lógica mais adequada, no entanto, não se deve basear apenas no

modelo conceitual, mas também levar em consideração os requisitos não funcionais da aplicação e as particularidades do BD alvo. Entre esses requisitos, destacam-se a consistência, a escalabilidade e o desempenho. É essencial refletir sobre como esses aspectos não funcionais serão atendidos pelo BD alvo, como nível de consistência exigido, transacional ou eventual, padrões de acesso e frequência com que os dados serão consultados ou modificados. Cada abordagem de modelagem lógica apresenta *trade-offs* que devem ser avaliados de acordo com o domínio do negócio e os requisitos da aplicação. Por isso, cada uma das oito abordagens propostas neste experimento foi analisada com base nesses critérios, seguindo as boas práticas de modelagem de agregados descritas por autores como Sadalage, Fowler, Eric Evans e Vaughn Vernon, as quais se alinham às diretrizes recomendadas pela equipe de desenvolvimento do Apache Cassandra. Por fim, a Tabela 4 resume as abordagens implementadas, facilitando a replicação do experimento em outros tipos de bancos *NoSQL*, como os orientados a chave-valor.

Tabela 4 – Abordagens lógicas do experimento.

Abordagem	**Descrição**
Tabela com *array* de elementos embutidos	A entidade *cliente* tem um *array* de *veículos* relacionados por valor (cópia).
Tabela com *array* de elementos embutidos e tabela redundante	A entidade *cliente* tem um *array* de *veículos* relacionados por valor (cópia) e com redundância parcial (elemento *veículo*). Ou seja, os *veículos* são armazenados também em outra tabela.
Tabela com *array* de elementos embutidos e tabela redundante com apenas um elemento embutido	A entidade *cliente* tem um *array* de *veículos* relacionados por valor (cópia) e com redundância total (elemento *veículo* e *cliente*). o agregado redundante conta com inversão da raiz do agregado de origem. Ou seja, os *veículos* com informações do seu dono (*cliente*) são armazenados também em outra tabela.
Tabela com apenas um elemento embutido	Inversão da raiz do agregado. A entidade *veículo* tem um relacionamento por valor (cópia) a *cliente*.
Tabela com apenas um elemento embutido e tabela redundante	Inversão da raiz do agregado com redundância. A entidade *veículo* tem um relacionamento por valor (cópia) ao *cliente* e com redundância parcial (elemento *cliente*). Ou seja, os *clientes* também são armazenados

	em outra tabela.
Cenário	**Descrição**
Tabela com *array* de elementos referenciados	A entidade *cliente* tem um *array* de *veículos* relacionados por referência. Ou seja, os *clientes* armazenam um *array* de identificadores de *veículos*.
Tabela com um elemento referenciado	Inversão do agregado com relacionamento por referência. A entidade *veículo* tem relacionamento por referência a *cliente*. Ou seja, os *veículos* armazenam identificador do *cliente*.
Tabela intermediária de referências	Criação de um agregado externo para relacionar as entidades *cliente* e *veículo* por referência. Ou seja, outra tabela que armazena os identificadores de *cliente* e *veículo*.

Fonte - Autoria própria.

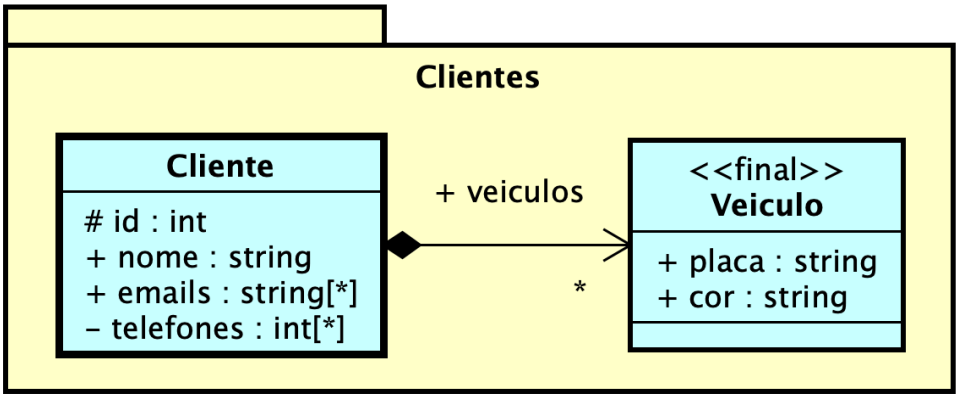
4.2. Experimento

4.2.1. Tabela com array de elementos embutidos

4.2.1.1. Modelagem lógica em AML

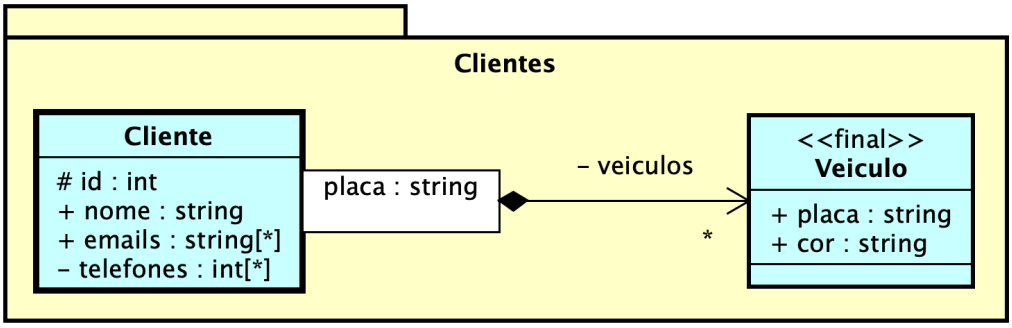
Na primeira abordagem analisada, adota-se um agregado *clientes*, composto pela entidade raiz *cliente* e o VO *veículo*, conectados por um *link* de composição com multiplicidade de "1:N", como ilustrado na Figura 40. Esse tipo de modelagem é útil em sistemas, em que apresentam invariantes de negócios transacionais entre *cliente* e *veículo* e o acesso do agregado é mais importante do que a manipulação individual dos seus componentes. No Apache Cassandra, há uma limitação técnica que exige que os UDTs inseridos em coleções (como listas) sejam definidos como *frozen*, o que implica que os objetos são tratados como objetos serializados e imutáveis, sem suporte a polimorfismo ou atualização parcial. Por esse motivo, na Figura 40, o VO *veículo* é representado na AML com o construtor *estereótipo final*, indicando que se trata de uma estrutura imutável, equivalente ao uso de *frozen<UDT>* no Cassandra. Nessa modelagem de lista embutida também é analisado o cenário apresentado na Figura 41, que utiliza um *link* de composição qualificada entre *cliente* e *veículo* com objetivo de avaliar a implementação desse conceito no Apache Cassandra por meio da estrutura *Map*.

Figura 40 - Modelagem lógica da abordagem da tabela com *array* de elementos embutidos.



Fonte - Autoria própria.

Figura 41 - Modelagem lógica da abordagem da tabela com *map* de elementos embutidos.



Fonte - Autoria própria.

A entidade raiz *cliente* e o VO *veículo* formam o agregado *clientes* com relacionamento de composição "1:N", em que os elementos são tratados como uma unidade lógica, ou seja, são armazenados, consultados e modificados em conjunto, garantindo atomicidade das operações. Por isso, essa abordagem se baseia no conceito de desnormalização, visando otimizar a leitura de dados relacionados que devem ser retornados juntos, além de permitir a aplicação de invariantes de negócio com consistência transacional local. O VO *veículo* depende da existência da entidade *cliente* e só pode ser acessado ou alterado por meio das regras (invariantes) da entidade *cliente*. Por exemplo, considere uma regra de negócio que restringe cada cliente a possuir no máximo três veículos ativos. Essa regra pode ser implementada diretamente no agregado, pois todos os veículos do cliente estão disponíveis localmente na mesma linha da tabela, permitindo que a aplicação valide a quantidade total antes de permitir a inserção de um novo item. Como os dados estão modelados em conjunto, a verificação e a aplicação da restrição ocorrem de forma transacional e consistente, sem depender de consultas distribuídas. A seguir, são descritas as propriedades e restrições envolvidas:

- Entidade *cliente*
  - *id* do tipo *int*: É o identificador único da entidade *cliente*, possui a restrição de unicidade.
  - *nome* do tipo *string*: Representa o nome do cliente;
  - *emails* do tipo *array* de *string*: Representa os e-mails associados ao cliente. Com

multiplicidade “N”.

- *telefones* do tipo *array* de *string*: Representa os telefones associados ao cliente que não podem repetir valor, com multiplicidade “N”, além de possuir essa restrição de unicidade.

- VO *veículo*
  - *placa* do tipo *string*: Representa a placa do veículo do cliente.
  - *cor* do tipo *string*: Representa a cor do veículo do cliente.

4.2.1.2. Aplicação no Apache Cassandra

No Apache Cassandra, essa modelagem é implementada por meio de uma tabela, representando o agregado que garante a consistência transacional por linha, uma vez que Cassandra assegura a atomicidade apenas no nível de partição. Para viabilizar o *link* de composição entre entidades e VOs, é necessário criar um UDT, que define uma estrutura personalizada para o elemento que será embutido. Na primeira abordagem, foi criado o UDT *veículo* para representar o VO *veículo*, conforme o comando demonstrado na Figura 42. Em seguida, a tabela *clientes* foi criada com base na modelagem lógica expressa em AML, conforme o *script* apresentado na Figura 43. Durante o processo de modelagem física, foram considerados os seguintes mapeamentos entre os elementos da AML e a estrutura do Cassandra:

- Atributos com pictograma de unicidade (identificador) na AML são mapeados como chave primária na tabela.
- Atributos com multiplicidade N e restrição de unicidade são mapeados para o tipo *set*.
- Atributos com multiplicidade N sem restrição de unicidade são mapeados para o tipo *list*.

Figura 42 - *Script* de CQL para criação do UDT *veículo*.

```
CREATE TYPE IF NOT EXISTS veiculo (
  placa text,
  cor text
);
```

Fonte - Autoria própria.

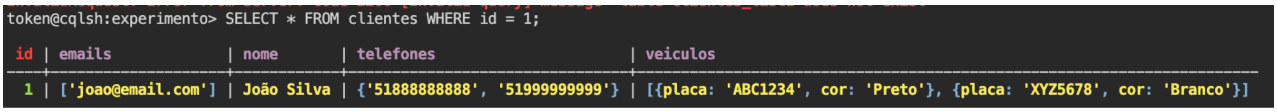
Figura 43 - *Script* de CQL para criação da tabela *clientes*.

```
CREATE TABLE IF NOT EXISTS clientes (
  id int PRIMARY KEY,
  nome text,
  emails list<text>,
  telefones set<text>,
  veiculos list<frozen <veiculo>>
);
```

Fonte - Autoria própria.

Com essa estrutura, é possível realizar consultas utilizando o *identificador do cliente* como filtro, retornando o agregado completo, conforme ilustrado na Figura 44. Caso seja necessário realizar consultas por atributos secundários, como o *nome do cliente*, é recomendável a configuração de um índice.

Figura 44 - Resultado de consulta por meio do comando *SELECT* do CQL na tabela *clientes*.



Fonte - Autoria própria.

Além disso, também é possível implementar a modelagem lógica com *link* de composição qualificada, que no contexto do Apache Cassandra é representada por meio de uma estrutura *map*, permitindo que os *veículos* sejam armazenados e acessados diretamente por uma chave identificadora (por exemplo, a placa). A criação dessa estrutura é demonstrada na Figura 45, e a Figura 46 apresenta um exemplo de consulta utilizando o comando *SELECT* para recuperar um veículo específico associado ao cliente com base na chave do mapa.

Figura 45 - *Script* de CQL para criação da tabela *clientes* com *map*.

```
CREATE TABLE IF NOT EXISTS clientes (
  id int PRIMARY KEY,
  nome text,
  emails list<text>,
  telefones set<text>,
  veiculos map<text, frozen<veiculo>>
);
```

Fonte - Autoria própria.

Figura 46 - Resultado de consulta por meio do comando *SELECT* do CQL na tabela *clientes* com *map*.

```
token@cqlsh:experimento> SELECT veiculos['ABC1234'] FROM clientes WHERE id = 1;

 veiculos['ABC1234']
-------------------------------------
 {placa: 'ABC1234', cor: 'Vermelho'}

(1 rows)
```

Fonte - Autoria própria.

4.2.1.3. Avaliação da abordagem

Vantagens

- **Atomicidade**: Como os dados da entidade e do VO são armazenados em uma única linha no Apache Cassandra, todas as alterações no agregado são realizadas de forma atômica, garantindo consistência local ao conjunto de dados e aplicação das invariantes de negócio.
- **Desempenho no acesso aos dados do *cliente***: Devido a desnormalização dos dados do *veículo*, a estrutura embutida no agregado *clientes* permite o acesso completo a entidade *clientes* e seus VOs *veículos* com uma única operação de leitura, ou seja, ler somente um nó, o que reduz a latência e o número de consultas necessárias.
- **Simplicidade na manutenção**: O uso de estrutura embutida simplifica a modelagem, pois reduz a necessidade de tabelas adicionais e de controle de integridade entre entidades.

Desvantagens

- **Tamanho do registro:** Como todos os dados do VO são embutidos, a linha pode crescer em volume, comprometendo boas práticas do Cassandra que recomendam evitar registros grandes.
- **Dificuldade de atualização das propriedades de *veículo*:** Caso seja necessário optar pela estrutura *frozen*, qualquer modificação em um item do VO exige a substituição da estrutura completa, o que pode ser custoso em termos de desempenho.
- **Problemas de consistência por concorrência**: Caso ocorra atualizações frequentes no objeto embutido e com acesso de múltiplos usuários ao mesmo tempo, o sistema enfrentará problemas de consistência por concorrência, pois o sistema altera versões mais antigas dos objetos.
- **Limitação nas consultas**: Não é possível realizar buscas diretas por atributos internos do VO (ex.: filtrar veículos pela cor), a menos que sejam duplicados na estrutura da tabela *cliente* ou em outra tabela redundante. Ou na criação de índices.

Consistência

Com essa abordagem é possível entregar uma consistência transacional local por meio da atomicidade devido ao VO e entidade serem armazenados juntos na mesma linha no Apache Cassandra. O VO é uma coluna de tipo customizado (*UDT*) da tabela *clientes*. Podem ocorrer problemas de consistência por concorrência, caso ocorra atualizações frequentes no objeto embutido.

Disponibilidade

Por serem armazenados na mesma linha, caso todos os *nodes* que armazenam as informações do *cliente* falharem, então as informações do *veículo* também estarão indisponíveis.

Tolerância à partição

Essa abordagem consegue escalar horizontalmente utilizando a chave de partição *id* do *cliente*. Além disso, a chave de partição utilizada nesta abordagem apresenta uma boa distribuição de dados por utilizar o identificador único do *cliente* que não gera *hotspots*. Caso a quantidade de *veículos*

por *cliente* cresça de forma indefinida e ilimitada além do recomendado pelo Apache Cassandra, isso representará problemas na replicação dos dados.

Desempenho

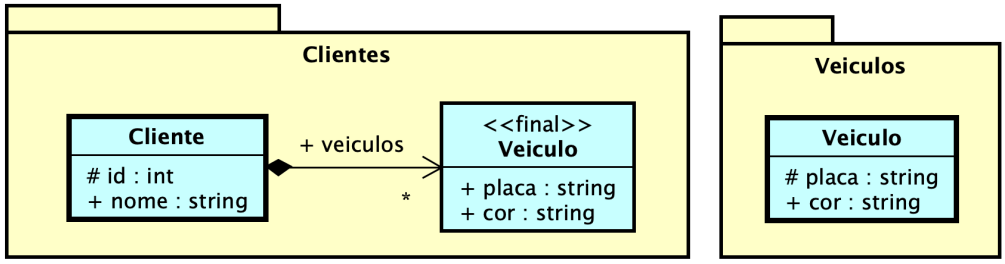
Bom desempenho na leitura e escrita em consultas que envolvem a chave de partição que seria a propriedade *id* do *cliente*, mas caso seja necessário consultas utilizando outras colunas da entidade *cliente* deverá ser necessário a criação de índices. O desempenho de escrita de itens embutidos são menos eficientes que estruturas normalizadas. E o desempenho pode ser degradado em cenários de escrita frequente com VO de grande tamanho ou se o conteúdo embutido crescer além do recomendado.

4.2.2. Tabela com array de elementos embutidos e tabela redundante

4.2.2.1. Modelagem lógica em AML

Na segunda abordagem analisada, apresentada na Figura 47, mantém-se a estrutura de *array* de elementos embutidos, conforme adotado na abordagem anterior. No entanto, essa proposta é complementada com a criação de um agregado redundante parcial, no qual os dados dos *veículos* são armazenados também em outro agregado apartado. Essa redundância parcial, visa trazer maior flexibilidade e melhoria na *performance* de consultas às informações de *veículo*, especialmente quando há necessidade de filtrar por seus atributos como *cor*, *modelo* ou *placa*. Porém, essa abordagem traz consigo o desafio em manter os dados consistentes entre as duas estruturas, uma vez que o Apache Cassandra não oferece suporte a transações entre múltiplas tabelas. Ao duplicar os dados do *veículo*, a consistência transacional deixa de ser garantida automaticamente, sendo necessária sua implementação pela lógica da aplicação. Para diminuir essa problemática, é possível utilizar o comando *BEGIN BATCH*, que permite agrupar operações de escrita em diferentes tabelas com garantia de atomicidade, embora ainda sem controle total de isolamento.

Figura 47 - Modelagem lógica da abordagem da tabela com *array* de elementos embutidos e tabela redundante.



Fonte - Autoria própria.

Nesse cenário, o agregado *clientes* mantém a entidade raiz *cliente* e o VO *veículo* mantendo o relacionamento de composição 1:N, em que os *veículos* continuam embutidos na tabela *clientes*. Em paralelo, é criado o agregado *veículos*, com estrutura independente, onde cada *veículo* é armazenado com restrição de unicidade sobre o atributo placa. Esse agregado é considerado redundante por conter os mesmos dados já existentes na tabela de *clientes*.

4.2.2.2. Aplicação no Apache Cassandra

Na segunda abordagem, a aplicação no Apache Cassandra se baseia em duas tabelas: a *clientes*, que mantém a estrutura com *array* de *veículos* embutidos, e a *veículos*, que atua como tabela redundante com os dados do *veículo* armazenados separadamente. A criação do UDT *veículo*, necessário para embutir os dados na tabela *clientes*, segue o mesmo padrão da abordagem anterior. A modelagem da tabela *clientes* também, utilizando a estrutura *list<frozen<veiculo>>* para representar a multiplicidade de *veículos*. A novidade nesta abordagem está na criação da tabela *veiculos*, apresentada na Figura 48, que viabiliza a realização de consultas independentes ao *veículo*, sem a necessidade de leitura da estrutura embutida no agregado *cliente*. Por exemplo, buscar *veículo* pela *placa*, conforme demonstrado na Figura 49.

Figura 48 - *Script* de CQL para criação da tabela *veículos*.

```
CREATE TABLE IF NOT EXISTS veiculos (
  placa text PRIMARY KEY,
  cor text
);
```

Fonte - Autoria própria.

Figura 49 - *Script* de CQL de consulta da tabela *veículos*.

```
token@cqlsh:experimento> SELECT * FROM veiculos WHERE placa = 'DEF1234';

 placa   | cor
---------+-------
 DEF1234 | Cinza

(1 rows)
```

Fonte - Autoria própria.

Com essa modelagem, consultas podem ser feitas tanto pela chave primária do *cliente* na tabela *clientes*, quanto pela *placa do veículo* na tabela *veículos*, como demonstrado na Figura 49. Essa estratégia proporciona maior flexibilidade de leitura, porém exige da aplicação o controle manual da consistência entre as duas estruturas.

4.2.2.3. Avaliação da abordagem

Vantagens

- **Boa *performance* na leitura de dados agregados e redundantes:**
    - Os dados dos *veículos* atrelados ao *cliente* ainda estão embutidos devido a desnormalização, com isso é possível realizar leitura rápidas de todo o agregado *cliente* com uma única consulta, e consequentemente, menor latência. Ou seja, caso necessário consultar o agregado *clientes* com a entidade *cliente* e seus *veículos*, só será necessário consultar uma tabela e realizar somente uma *query*.
    - E pelos os dados de *veículos* estarem redundantes em outra tabela, então é possível realizar a consulta mais performática quando necessário somente ler informações dos *veículos* por meio de sua chave.

- **Flexibilidade de consulta:** A tabela redundante permite consultas filtradas diretamente por atributos do VO, como *cor*, *placa* ou qualquer outro campo, sem a necessidade de varrer todo o agregado.

Desvantagens

- **Risco de inconsistências:** Por não haver implementar estrutura de transações no Cassandra, a consistência entre as duas tabelas depende da aplicação, que deve coordenar as inserções, atualizações e exclusões corretamente de forma atômicas utilizando estruturas de *BEGIN BATCH* por exemplo.
- **Maior complexidade de escrita:** A aplicação deve garantir ao atualizar o registro embutido Veículo também atualizar a tabela redundante para manter a consistência.
- **Consistência eventual:** A aplicação das alterações entre o registro embutido e a tabela redundante será realizada de forma eventual e não atômica. Uma das formas de diminuir essa problemática é a utilização de *BEGIN BATCH* do Apache Cassandra para manter a operação atômica.
- **Aumento no armazenamento por redundância:** Os dados dos *veículos* são armazenados em dois locais, o que aumenta o espaço em disco;
- **Maior latência na escrita:** As operações de escrita tornam-se mais pesadas, pois devem ser realizadas em múltiplas tabelas para manter a sincronização dos dados e consistência.

Consistência

A atomicidade só é garantida dentro do agregado embutido (na tabela *clientes*), porém devido a redundância dos dados em outra tabela, a consistência entre diferentes agregados é eventual, e dependente da lógica da aplicação, que precisa controlar a replicação dos dados entre as estruturas.

Disponibilidade

Aumenta a disponibilidade, pois caso caso todos os *nodes* que armazenam as informações do *cliente* falharem, pode ser que os *nodes* com as informações do *veículo* ainda estejam disponíveis.

Tolerância à partição

Essa abordagem ainda permite escalar bem, principalmente, a tabela redundante e também a tabela com embutido, caso o registro embutido não cresça além do recomendado.

Desempenho

Em relação ao desempenho, essa abordagem apresenta leituras otimizadas, tanto para o acesso ao agregado completo — por meio da tabela *clientes* com os *veículos* embutidos — quanto para consultas específicas ao VO, realizadas diretamente na tabela redundante. No entanto, as operações de escrita tendem a ser mais custosas, uma vez que exigem atualizações duplicadas em ambas as tabelas, além de um controle transacional externo na aplicação para garantir a consistência entre elas. Quanto ao armazenamento, o uso de dados redundantes aumenta o consumo de espaço em disco, especialmente em cenários com agregados que possuem grande volume de objetos do tipo Veículo.

4.2.3. Tabela com array de elementos embutidos e tabela redundante com apenas um elemento embutido

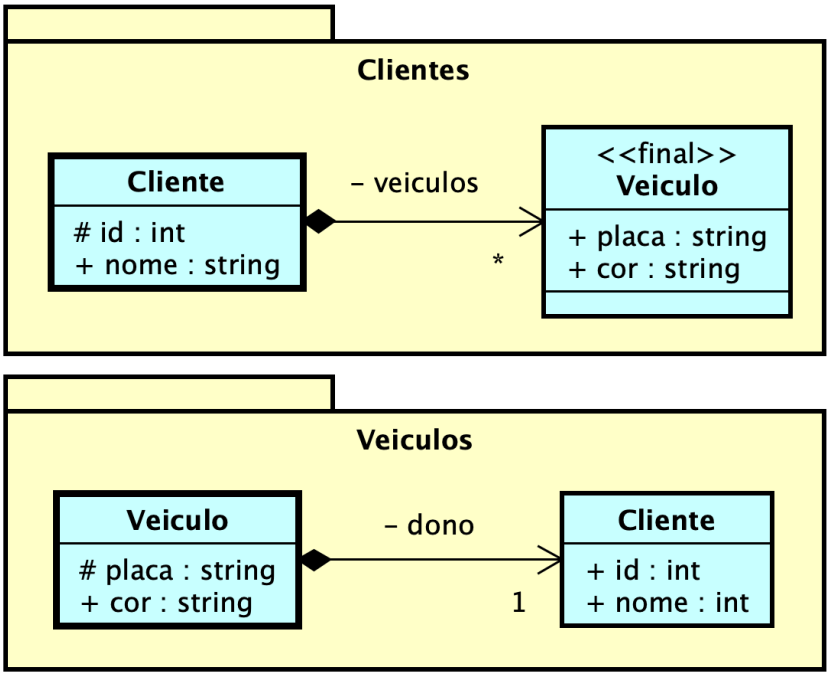
4.2.3.1. Modelagem lógica em AML

Na terceira abordagem analisada, apresentada na Figura 50, mantém-se a estrutura do agregado *clientes* com *array* de elementos embutidos de VO *veículo*, como nas abordagens anteriores. Porém, nesta abordagem, a redundância é realizada de forma completa, com a criação do agregado *veículos*, no qual o *veículo* assume como a entidade raiz e o *cliente* como um VO embutido. Com isso, no agregado redundante há uma inversão total da estrutura e da raiz do agregado em relação ao modelo principal, além da mudança na cardinalidade do relacionamento, que passa de “1:N” para “1:1”.

Esse modelo apresenta um grau mais elevado de desnormalização em comparação às demais abordagens do experimento, ao replicar integralmente os dados entre dois agregados distintos. O resultado é a criação de uma estrutura capaz de retornar dados completos tanto no agregado *clientes* quanto no agregado *veículos* com maior flexibilidade nas consultas. Essa abordagem é vantajosa em aplicações que exigem buscas distintas e direcionadas por entidade, como consultas por todos os *veículos* de um *cliente*, com base em filtros sobre o *cliente*, ou por um *veículo* específico, incluindo

os dados do proprietário, a partir de filtros aplicados diretamente ao *veículo*. Por outro lado, essa replicação total acarreta maiores desafios relacionados à consistência dos dados. À medida que a redundância aumenta entre diferentes tabelas, também cresce a possibilidade de divergência entre os dados. Como o Apache Cassandra não oferece suporte a transações entre múltiplas tabelas, a consistência entre os agregados deve ser gerenciada pela lógica da aplicação, aumentando a complexidade e o risco de inconsistência, especialmente em cenários com atualizações concorrentes ou de alta frequência.

Figura 50 - Modelagem lógica da abordagem da tabela com *array* de elementos embutidos e tabela redundante com apenas um elemento embutido.



Fonte - Autoria própria.

No agregado principal, a entidade *cliente* armazena os *veículos* como uma lista embutida, com relacionamento de composição "1:N". Já na tabela redundante, o *veículo* é a entidade raiz de um novo agregado, e o *cliente* aparece como um VO embutido, com relacionamento "1:1", modelado por composição.

4.2.3.2. Aplicação no Apache Cassandra

Na terceira abordagem, a aplicação no Apache Cassandra reutiliza a estrutura da tabela *clientes*

definida nas abordagens anteriores, mantendo o VO *veículo* embutido como uma lista de *frozen<veiculo>*. A principal diferença está na modelagem da tabela redundante, que neste caso representa uma redundância total, com a entidade *veículo* a raiz, e o *cliente* é armazenado como um VO embutido por meio de um relacionamento de composição 1:1. Para representar essa estrutura, foi criado um novo UDT chamado *cliente*, utilizado como campo embutido na tabela *veículos*, conforme demonstrado na Figura 51. Como neste caso o cliente não está dentro de uma lista ou coleção, não é obrigatório o uso de *frozen*, embora ele ainda possa ser aplicado quando necessário para manter a integridade do valor como uma unidade indivisível.

Figura 51 - *Script* de CQL para criação do UDT *cliente*.

```
CREATE TYPE IF NOT EXISTS cliente (
  id int,
  nome text
);
```

Fonte - Autoria própria.

A tabela *veículos* é então criada com um campo do tipo *cliente*, embutido diretamente no registro, conforme ilustrado na Figura 52. Essa modelagem permite que os dados do *cliente* sejam acessados diretamente ao consultar um *veículo*, sem necessidade de múltiplas leituras ou junções entre tabelas, otimizando o desempenho de leitura orientada ao *veículo*.

Figura 52 - *Script* de CQL para criação da tabela *veículos* com redundância total.

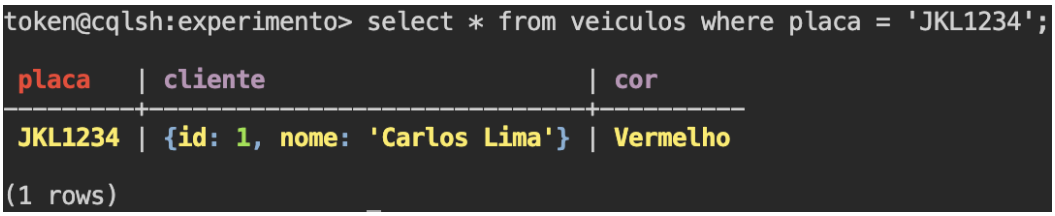
```
CREATE TABLE IF NOT EXISTS veiculos (
  placa text PRIMARY KEY,
  cor text,
  cliente cliente
);
```

Fonte - Autoria própria.

Com essa estrutura, é possível realizar consultas performáticas por *placa*, retornando tanto os dados do *veículo* quanto os dados do *cliente* embutido, conforme demonstrado na Figura 53. Além disso, ainda é possível utilizar a tabela *clientes* para realizar consultas por *id do cliente*, retornando o agregado completo com os *veículos* embutidos, como nas abordagens anteriores.

Dessa forma, esta abordagem é vantajosa em cenários nos quais necessita de acesso a todo o agregado e pode ocorrer tanto a partir da entidade *veículo* quanto da entidade *cliente*.

Figura 53 - *Script* de CQL de consulta da tabela *veículos* com redundância total.



Fonte - Autoria própria.

4.2.3.3. Avaliação da abordagem

Vantagens

- **Flexibilidade nas consultas:**
    - Com *veículo* como entidade raiz, é possível realizar consultas específicas pelo atributo *placa*, ou até por meio de índices utilizando a coluna *cor*, de maneira mais eficiente e carregando o dono (*cliente* associado) a ele. Com isso, atendendo as consultas que necessitam das informações do *veículo* e do seu dono (*cliente* associado).
    - Além de ser possível também realizar consultas para retornar os *clientes* e seus *veículos* no agregado *clientes* utilizando *id do cliente*, ou outras colunas por meio de índices quando necessário.
- ***Performance* na consulta de informações**: Devido a tabela redundante e da desnormalização de dados do *cliente* e *veículo*, é possível realizar consultas performáticas por meio da chave primária e de partição placa para buscar informações do *veículo* e do seu proprietário. Além de conseguir consultar os dados dos *clientes* com seus *veículos* associados também por meio da tabela de *clientes*.

Desvantagens

- **Risco de inconsistências:** O risco de inconsistências aumenta devido a necessidade de realizar a sincronização de dados em duas tabelas com estruturas desnormalizadas em ambas as tabelas. Além disso, não há garantia de consistência transacional no Cassandra, a

consistência entre as duas tabelas depende da aplicação, que deve coordenar as inserções, atualizações e exclusões corretamente.

- **Consistência eventual:** Assim, como no cenário anterior, a aplicação das alterações entre o registro embutido e a tabela redundante é de forma eventual e não atômica.
- **Maior complexidade de escrita:** A aplicação deve garantir a atualizações as tabelas redundantes quando o registro embutido é alterado e vice-versa para manter a consistência
- **Redundância dos dados:** Os dados dos *veículos* e *clientes* são armazenados em dois locais com armazenamento de estruturas embutidas redundantes em ambos os locais, o que aumenta o espaço em disco;
- **Maior latência na escrita:** As operações de escrita tornam-se mais pesadas, pois devem ser realizadas em múltiplas tabelas para manter a sincronização.

Consistência

Devido a redundância dos dados entre as tabelas, a consistência entre esses diferentes agregados é eventual, e a aplicação deve implementar uma regra de atomicidade para replicar a alteração dos dados entre as tabelas que tem redundância.

Disponibilidade

Ambas as estruturas operam de forma independente, e a indisponibilidade de uma delas não compromete a outra. Isso traz benefícios para a disponibilidade geral do sistema, principalmente em cenários onde agregados são lidos de forma distinta.

Tolerância à partição

Precisa ter cuidado para que as tabelas com estruturas embutidas não cresçam além do recomendado para não comprometer a escalabilidade horizontal.

Desempenho

Em relação ao desempenho, essa abordagem apresenta leituras otimizadas que retornam de forma completa os dados das entidades. Desde consultas para buscar informações completas dos *clientes* com seus *veículos* até consultas de informações do *veículo* e do seu dono (*cliente* associado). No entanto, as operações de escrita são bem mais custosas, uma vez que exigem atualizações

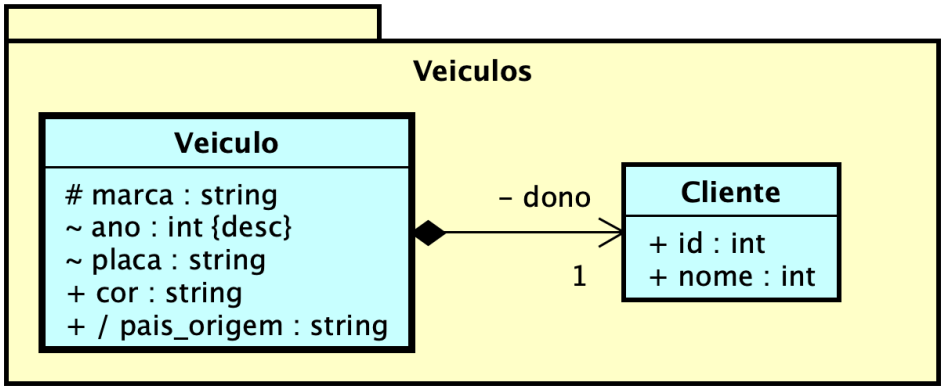
duplicadas em ambas as tabelas, além de um controle externo na aplicação para garantir a consistência entre elas. Quanto ao armazenamento, o uso de dados redundantes aumenta o consumo de espaço em disco, especialmente em cenários com agregados que possuem grande volume de objetos do tipo *veículo* e *cliente*.

4.2.4. Tabela com apenas um elemento embutido

4.2.4.1. Modelagem lógica em AML

Na quarta abordagem analisada, apresentada na Figura 54, ocorre uma inversão da estrutura do agregado em relação ao modelo original. Nesta proposta, a entidade *veículo* passa a ser a raiz do agregado, enquanto *cliente* é modelado como um VO embutido em um relacionamento *link* de composição 1:1. Diferentemente da terceira abordagem, esta não apresenta redundância, ou seja, trata-se de um único agregado centrado em *veículo*, com os dados essenciais do *cliente* incorporados diretamente. Essa modelagem é útil em contextos em que existem regras invariantes a serem aplicadas entre *veículo* e *cliente* e as operações mais frequentes envolvem acesso direto ao *veículo*, e seu proprietário (*cliente* associado) é apenas um dado complementar

Figura 54 - Modelagem lógica da abordagem da tabela com apenas um elemento embutido.



Fonte - Autoria própria.

Nesse cenário, a entidade *veículo* torna-se o ponto de entrada do agregado e armazena os dados do *cliente* de forma embutida. O relacionamento entre eles é de composição 1:1, e os dados são tratados como uma unidade lógica, ou seja, são armazenados e consultados juntos, de forma atômica. Além disso, para ilustrar os conceitos de ordenamento e propriedade estática da AML e do Apache Cassandra, foram adicionadas três colunas à modelagem da entidade *veículo*: *marca*, *ano* e

país de origem. A coluna *marca* recebeu o pictograma discriminador que indica que é a chave de partição da tabela. Já a coluna *ano* recebe pictograma de discriminador e ordenamento {*desc*}, indicando que ela atua como *clustering key* ordenado por ordem decrescente. E a coluna *placa* também recebendo o pictograma discriminador para identificar que faz parte da chave primária. Já a coluna *país de origem* foi modelada como uma coluna estática, o que significa, no contexto do Cassandra, que ela possui o mesmo valor para todos os registros da mesma partição, sendo armazenada uma única vez por partição.

4.2.4.2. Aplicação no Apache Cassandra

Na quarta abordagem, a modelagem é realizada por meio de uma única tabela *veículos*, na qual *veículo* é a entidade raiz do agregado e o *cliente* é embutido como um VO em um relacionamento de composição 1:1. Para representar essa estrutura no Apache Cassandra, mantém-se a utilização do tipo personalizado *cliente*, já definido em abordagens anteriores, e a tabela *veículos* é ajustada para incorporar características adicionais da modelagem lógica. A coluna marca é utilizada como chave de partição, permitindo distribuir os dados entre os nós do *cluster* com base na fabricante do *veículo*. A ordenação dos registros dentro da partição é controlada pelas colunas *ano* e *placa*, que compõem a *clustering key*, sendo que o ano é definido em ordem decrescente, permitindo recuperar os modelos mais recentes primeiro. Além disso, a coluna *pais_origem* é definida como estática, armazenando um valor comum para todos os *veículos* de uma mesma marca, o que evita a duplicação de dados para esse tipo de informação. A estrutura é apresentada na Figura 55.

Figura 55 - *Script* de CQL de criação da tabela *veículos* com *clustering keys* e coluna estática.

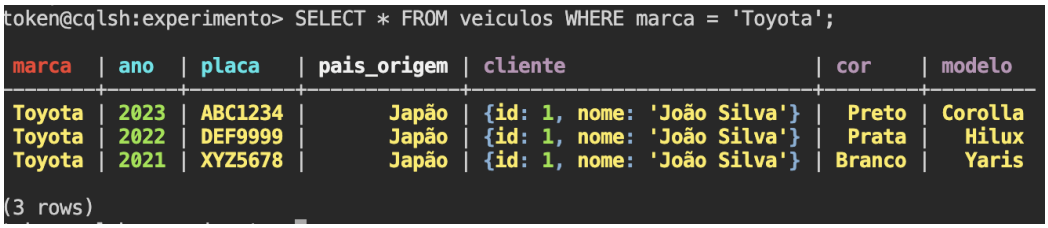
```
CREATE TABLE IF NOT EXISTS veiculos (
  marca text,
  ano int,
  placa text,
  modelo text,
  cor text,
  pais_origem text STATIC,
  cliente cliente,
  PRIMARY KEY (marca, ano, placa)
) WITH CLUSTERING ORDER BY (ano DESC, placa ASC);
```

Fonte - Autoria própria.

Essa modelagem permite realizar consultas performáticas por *marca*, retornando os *veículos*

mais recentes no topo dos resultados, junto com os dados do *cliente* embutido e o *país de origem*. Um exemplo de consulta é apresentado na Figura 56, que utiliza apenas a chave de partição (*marca*) e se beneficia da ordenação por ano.

Figura 56 - *Script* de CQL de consulta da tabela *veículos* com *clustering keys* e coluna estática.



Fonte - Autoria própria.

Essa abordagem é recomendada para cenários em que o acesso principal ocorre a partir da entidade *veículo*, e há interesse em visualizar os modelos mais novos primeiro, com informações complementares do *cliente* e dados de origem compartilhados por *marca*.

4.2.4.3. Avaliação da abordagem

Vantagens

- **Consultas otimizadas ao agregado *veículo*:** Como os dados do *cliente* estão embutidos diretamente no *veículo* devido a desnormalização dos dados, é possível obter todas as informações necessárias em uma única consulta, ideal para sistemas em que o foco está no *veículo*.
- ***Performance* na consulta de informações de *veículos***: Devido a desnormalização de dados do *cliente*, é possível realizar consultas performáticas com menor latência por meio da chave primária e de partição placa para buscar informações do *veículo* e do seu dono (*cliente* associado), sem precisar navegar por listas ou consultar múltiplas tabelas..
- **Simplicidade da estrutura:** A modelagem requer apenas uma tabela, sem necessidade de sincronização ou controle entre agregados distintos ou tabelas auxiliares.

Desvantagens

- **Perda da centralidade do *cliente* como raiz do agregado:** A modelagem deixa de refletir a estrutura natural do domínio, o que pode dificultar a manutenção ou expandir a complexidade quando for necessário trabalhar a partir do *cliente*.

- **Dificuldade na consulta do *cliente* para seus *veículos*:** Para obter todos os *veículos* de um *cliente* sem varrer toda a tabela de *veículos*, só é possível com criação de índices.
- **Baixa flexibilidade de consulta:** Não é possível realizar buscas diretas por atributos internos do VO (ex.: filtrar por *identificador do cliente*), a menos que sejam duplicados na estrutura da tabela de *veículos*, em outra tabela, ou criando índices no Apache Cassandra.
- **Aumento do acoplamento entre os dados do *cliente* e o agregado *veículo*:** Qualquer alteração no VO *cliente* embutido (ex.: mudança de *nome*) exige a atualização de todos os *veículos* relacionados, o que pode gerar inconsistência se não for bem controlado.

Consistência

Assim como na primeira abordagem pode ser entregue consistência transacional local por meio da atomicidade no Apache Cassandra.

Disponibilidade

Por serem armazenados na mesma linha, caso todos os *nodes* que armazenam as informações do *veículo* falham, então as informações do *cliente* também estarão indisponíveis.

Tolerância à partição

Essa abordagem consegue escalar bem horizontalmente utilizando a chave de partição *placa*, caso o registro não cresça além do recomendado. Além disso, a chave de partição utilizada nesta abordagem apresenta uma boa distribuição de dados por utilizar a placa para não gerar *hotspots*.

Desempenho

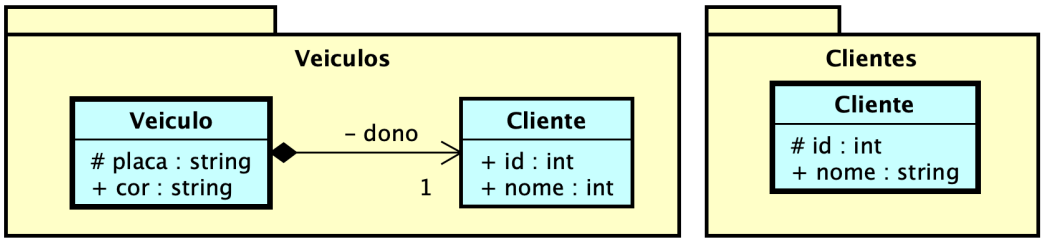
A *performance* de leitura e escrita é muito boa em consultas que envolvem a chave de partição que seria a propriedade *placa*, mas caso seja necessário consultas utilizando outras colunas da entidade *veículo* deverá ser necessário a criação de índices. O desempenho de escrita de itens embutidos são menos eficientes que estruturas normalizadas. E o desempenho pode ser degradado em cenários de escrita frequente com VO de grande tamanho ou se o conteúdo embutido crescer além do recomendado.

4.2.5. Tabela com apenas um elemento embutido e tabela redundante

4.2.5.1. Modelagem lógica em AML

A quinta abordagem analisada, apresentada na Figura 57, se assemelha à quarta abordagem que também ocorre uma inversão da raiz do agregado em relação ao modelo original, porém introduz no agregado redundante a redundância parcial. O agregado *veículos* conta com a entidade *veículo* como raiz e o elemento embutido VO *cliente*, e o agregado *clientes* que aplica redundância sobre os valores do VO *cliente*. O objetivo dessa abordagem é unificar os benefícios das leituras especializadas e completas, mantendo a flexibilidade de acesso direto ao *cliente*, sem comprometer a eficiência na leitura agregada a partir do *veículo*.

Figura 57 - Modelagem lógica da abordagem da tabela com apenas um elemento embutido e tabela redundante.



Fonte - Autoria própria.

No agregado *veículos*, o *veículo* é a entidade raiz e contém embutido um VO *cliente*, com relacionamento de composição 1:1. Esse agregado é armazenado na tabela *veiculos*, estruturado para leitura rápida do conjunto completo de *veículo* e de seu dono (*cliente* associado). Já na tabela redundante *clientes*, consta somente informações do *cliente*.

4.2.5.2. Aplicação no Apache Cassandra

Na quinta abordagem, mantém-se a tabela *veículos* com o *cliente* embutido como VO, conforme definido nas abordagens anteriores. A diferença nesta estrutura está na criação de uma tabela redundante parcial, na qual o *cliente* é representado como entidade raiz, mas sem armazenar os dados dos *veículos*. Essa modelagem permite a recuperação direta dos dados do *cliente* quando a leitura não exige informações dos *veículos*, reduzindo a complexidade e o volume de leitura em determinados cenários. A tabela *clientes* é criada com os atributos essenciais do *cliente*, como *id*, *nome*, *emails* e *telefones*, representando um agregado independente, conforme ilustrado na Figura

58. A redundância ocorre porque os dados do *cliente* também estão presentes, embutidos, na tabela *veiculos*.

Figura 58 - *Script* de CQL para criação da tabela *cliente* com redundância parcial.

```
CREATE TABLE IF NOT EXISTS clientes (
  id int PRIMARY KEY,
  nome text
);
```

Fonte - Autoria própria.

Com essa estrutura, é possível realizar consultas diretas por *id* na tabela *clientes* para recuperar apenas os dados do *cliente*, como demonstrado na Figura 59. Em paralelo, a tabela *veículos* pode ser utilizada para acessar os dados completos do *veículo* com o *cliente* embutido, conforme demonstrado nas abordagens anteriores. Esta abordagem oferece flexibilidade de leitura com menor custo para acessos simples, embora exija controle da aplicação para garantir a consistência entre os dados duplicados.

Figura 59 - *Script* de CQL de consulta da tabela *cliente* com redundância parcial.

```
token@cqlsh:experimento> select * from clientes where id = 1;

 id | nome
----+------------
  1 | João Silva

(1 rows)
```

Fonte - Autoria própria.

1.1.1.1. Avaliação da abordagem

Vantagens

- **Alta flexibilidade nas consultas:** A modelagem permite consultas otimizadas para informações do *cliente*, ao mesmo tempo que possibilita a leitura completa do agregado *veículos* com seus donos (*cliente* associado) em uma única consulta.
- **Independência entre agregados:** As duas estruturas são independentes e otimizadas para para diferentes necessidades de consultas. Caso seja necessário ler somente as informações do *cliente* ou leitura das informações do *veículo* com informações do seu dono.

- ***Performance* na consulta de informações de *veículos* e *clientes***: Caso necessário ler informações do *veículo* e do seu dono (*cliente* associado), devido a desnormalização de dados do *cliente*, é possível realizar consultas performáticas por meio da chave primária e de partição *placa*, sem precisar consultar múltiplas tabelas. E caso necessário ler somente informações da estrutura embutida *cliente*, consultar a tabela de *clientes* por meio do *identificador do cliente*.

Desvantagens

- **Redundância de dados:** Todos os dados de *cliente* são armazenados em duas tabelas diferentes, o que aumenta o uso de disco e a complexidade da manutenção.
- **Risco de inconsistência:** Qualquer alteração em um *cliente* precisa ser refletida em ambas as tabelas, o que exige controle rigoroso da aplicação para manter a consistência.
- **Aumento na latência de escrita:** Cada operação de escrita ou atualização requer pelo menos duas escritas sincronizadas.
- **Dificuldade na consulta do *cliente* para seus *veículos*:** Não é possível obter todos os *veículos* de um *cliente* sem varrer toda a tabela de *veículos*.
- **Aumento do acoplamento entre os dados do *cliente* e o agregado *veículo*:** Qualquer alteração no VO *cliente* embutido (ex.: mudança de *nome*) exige a atualização de todos os *veículos* relacionados e na tabela redundante, o que pode gerar inconsistência se não for bem controlado.
- **Baixa flexibilidade de consulta:** Não é possível realizar buscas diretas por atributos internos do VO (ex.: filtrar *veículos* pela *cor*), a menos que sejam duplicados na estrutura da tabela *cliente* ou em outra tabela redundante. Ou na criação de índices.

Consistência

A consistência entre o agregado *veículos* e *clientes* é eventual, exigindo que a lógica de negócio garanta a sincronia dos dados.

Disponibilidade

As duas estruturas operam de forma independente. Assim, mesmo que uma das tabelas esteja temporariamente indisponível, a outra pode continuar atendendo seu fluxo normalmente, aumentando a resiliência operacional.

Tolerância à partição

Essa abordagem consegue escalar bem horizontalmente utilizando a chave de partição *placa*, caso o registro não cresça além do recomendado. Por ter uma estrutura embutida com somente um valor, o registro não vai crescer além do recomendado.

Desempenho

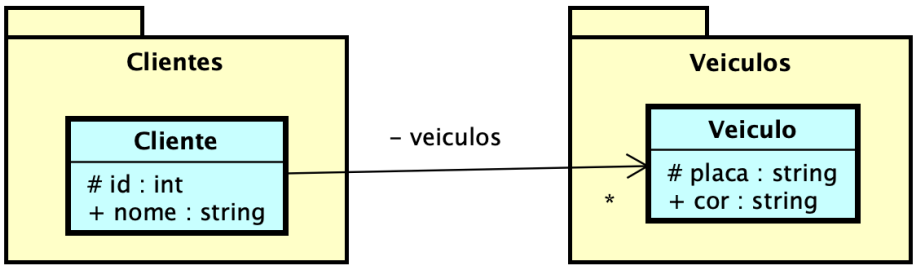
O desempenho de leitura é excelente tanto para consulta de um *cliente* específico quanto para consulta de um *veículo* específico com seu dono. Contudo, o desempenho de escrita é mais impactado, pois exige operações sincronizadas em duas tabelas. O armazenamento também é duplicado, o que pode ser relevante em cenários com grande volume de dados.

4.2.6. Tabela com array de elementos referenciados

4.2.6.1. Modelagem lógica em AML

Na sexta abordagem analisada, representada na Figura 60, é adotado um modelo onde a entidade *cliente* armazena um *array* de identificadores que fazem referência aos *veículos* associados a ele. Diferentemente das abordagens anteriores, que utilizavam composição com elementos embutidos (UDTs), esta abordagem adota um relacionamento por associação “1:N”, em que o *cliente* mantém apenas os identificadores dos *veículos*, e os detalhes dos *veículos* são armazenados separadamente em outra tabela. O objetivo desta modelagem é reduzir a duplicação de dados, permitindo acesso individualizado, eficiente e mais escalável aos *veículos*. Porém, caso necessário consultar informações das duas tabelas, será necessário realizar duas consultas que acarretam no aumento da latência nas consultas. Além disso, o Apache Cassandra não oferece suporte a restrições de integridade referencial, a consistência entre as tabelas depende da aplicação, que deve assegurar que as referências armazenadas em *clientes* realmente correspondam a registros existentes em *veículos*. Caso contrário, há risco de problemas como referências órfãs (ex.: deletar um *veículo* que ainda está vinculado a um *cliente*), comprometendo a integridade lógica dos dados.

Figura 60 - Modelagem lógica da abordagem da tabela com *array* de elementos referenciados.



Fonte - Autoria própria.

Na tabela *clientes*, temos a entidade raiz *cliente*, com um atributo *veiculos_ids*, representado como um *array* de *string*, que armazena as *placas dos veículos* (ou outro identificador único). A tabela *veículos* armazena os dados completos de cada *veículo*, com sua própria estrutura independente.

4.2.6.2. Aplicação no Apache Cassandra

Na sexta abordagem, a modelagem lógica adota o uso de referência entre agregados, evitando o armazenamento embutido dos dados. A entidade *cliente* mantém um *array* de identificadores de *veículos*, formando um relacionamento de associação 1:N por referência. Os dados dos *veículos* são mantidos em uma tabela separada, denominada *veículos*. Para implementar essa estrutura no Apache Cassandra, a tabela *clientes* é criada com uma coluna *veículos* do tipo *set<text>*, onde são armazenadas as *placas dos veículos* associados ao *cliente*, conforme apresentado na Figura 61. A escolha por *set* garante que não haja duplicação de identificadores, já que cada placa deve ser única dentro da coleção. Os dados completos de cada *veículo* permanecem na tabela *veículos*, conforme ilustrado na Figura 62, criada separadamente, com placa como chave primária.

Figura 61 - *Script* de CQL para criação da tabela *cliente* com *array* de identificadores de *veículo*.

```
CREATE TABLE IF NOT EXISTS clientes (
  id int PRIMARY KEY,
  nome text,
  veiculos_ids set<text>
);
```

Fonte - Autoria própria.

Figura 62 - *Script* de CQL para criação da tabela *veículo* sem elementos embutidos e referências.

```
CREATE TABLE IF NOT EXISTS veiculos (
  placa text PRIMARY KEY,
  cor text
);
```

Fonte - Autoria própria.

Essa abordagem oferece separação entre as entidades, menor redundância, maior escalabilidade e facilita o gerenciamento individual dos *veículos*. No entanto, como o Cassandra não oferece suporte a *joins* para recuperar os dados completos, é necessário realizar duas consultas: uma na tabela *clientes* para obter os identificadores dos *veículos*, e em seguida outra(s) na tabela *veículos*, como demonstrado na Figura 63.

Figura 63 - *Script* de CQL para consulta nas tabelas *cliente* e *veículo*.

```
token@cqlsh:experimento> SELECT * FROM clientes WHERE id = 1;

 id | nome             | veiculos_ids
----+------------------+------------------------
  1 | Fernanda Ribeiro | {'ABC1234', 'XYZ5678'}

(1 rows)
token@cqlsh:experimento> SELECT * FROM veiculos WHERE placa = 'ABC1234';

 placa   | cor
---------+-------
 ABC1234 | Preto

(1 rows)
```

Fonte - Autoria própria.

Essa modelagem é adequada para aplicações em que os dados do *cliente* e dos *veículos* são manipulados separadamente e o acoplamento entre as entidades pode ser leve. Por outro lado, a consistência entre as tabelas deve ser garantida pela aplicação, pois o Apache Cassandra não aplica restrição referencial, especialmente para evitar referências órfãs em cenários de exclusão ou atualização.

4.2.6.3. Avaliação da abordagem

Vantagens

- **Redução da duplicação de dados:** Os dados dos *veículos* são armazenados em apenas um local, e não são embutidos nem duplicados, o que reduz o uso de armazenamento.

- **Flexibilidade para acesso individual:** O acesso a um *veículo* específico é feito de forma direta na tabela *veiculos*, sem carregar estruturas maiores, ideal para consultas específicas ou manutenção de *veículos*.
- **Separação de responsabilidade entre entidades**: *Cliente* e *veículo* são modelados como agregados independentes, o que está em conformidade com o DDD que indica que sempre que possível criar agregados menores, quando há operações distintas e independentes sobre cada entidade.
- **Maior escalabilidade:** As entidades *cliente* e *veículos* por estarem em agregados distintos cada um será particionado entre os nós do *cluster* de forma apartada, e incentivando a distribuição dos dados. Com isso, essa abordagem cria registros menores que conseguem ser facilmente escaláveis.

Desvantagens

- **Perda de atomicidade:** Como os dados estão em tabelas separadas, não há garantia de consistência transacional entre a entidade *cliente* e seus *veículos*. As operações precisam ser coordenadas pela aplicação.
- **Maior complexidade de leitura do relacionamento:** Para recuperar todos os veículos de um cliente, é necessário realizar múltiplas leituras individuais na tabela Veículos, uma para cada identificador, o que pode resultar em maior latência e *overhead*.
- **Ausência de integridade referencial:** Apache Cassandra não possui suporte a chaves estrangeiras, logo, a aplicação deve garantir que os identificadores listados no *cliente* realmente existam na tabela de *veículos*.
- **Risco de inconsistência em exclusões e atualizações:** Se um *veículo* for removido da tabela *veiculos*, seu identificador pode permanecer na lista do *cliente*, gerando uma referência inválida que precisa ser controlada manualmente.

Consistência

Na separação entre os agregados distintos de *clientes* e *veículos*, a consistência torna-se eventual. Com isso, a sincronização e atomicidade entre os dois agregados depende da aplicação.

Disponibilidade

As tabelas **clientes** e *veículos* operam de forma independente. A disponibilidade do agregado como um todo depende da disponibilidade das duas tabelas, especialmente nas leituras combinadas. Entretanto, cada entidade pode ser acessada de forma isolada em casos de falha parcial.

Tolerância à partição

Essa abordagem apresenta alta tolerância à partição, pois cada um dos agregados são distribuídos de forma independente. Além de manter o tamanho dos registros de acordo com o recomendado pelo Apache Cassandra.

Desempenho

A *performance* de leitura de um único veículo e cliente é mais eficiente, porém quando se refere ao agregado completo (*cliente* + seus *veículos*) é inferior às abordagens com dados embutidos, pois exige múltiplas leituras individuais, uma para cada identificador. A escrita também é otimizada, pois cada entidade é atualizada separadamente, sem necessidade de reprocessar estruturas compostas.

4.2.7. Tabela com um elemento referenciado

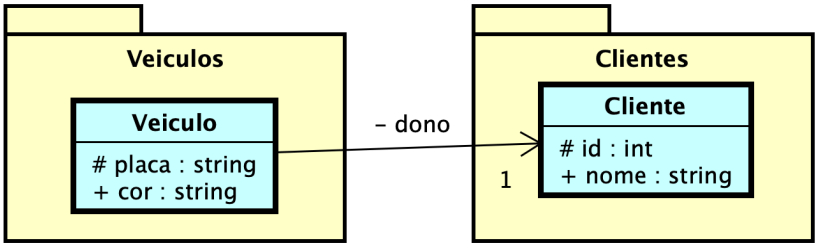
4.2.7.1. Modelagem lógica em AML

Na sétima abordagem, representada na Figura 64, adota-se uma estrutura em que o agregado *veículos* mantém uma referência ao agregado *clientes*, sem ser embutido como VO. O relacionamento entre *veículo* e *cliente* é de associação “1:1”, com a entidade *veículo* mantendo apenas uma referência ao *cliente*, geralmente por meio do *id_cliente*. Nesse modelo, os dados do *cliente* são armazenados exclusivamente em sua própria tabela, representando um agregado independente, o que reduz a redundância e permite maior independência entre as entidades.

Essa abordagem se diferencia da anterior, na qual era o *cliente* que mantinha um *array* de identificadores dos *veículos*. Aqui ocorre o inverso, onde o *veículo* que referencia seu proprietário. Essa configuração é adequada para cenários em que as operações são iniciadas a partir da entidade *veículo*, nos quais as consultas são feitas primeiramente sobre o *veículo* e, em alguns casos, os dados do *cliente* são complementares. Assim, quando necessário obter ambos, a

aplicação pode consultar o *veículo* e, em seguida, realizar uma leitura complementar do *cliente* por meio do identificador armazenado.

Figura 64 - Modelagem lógica da abordagem da tabela com um elemento referenciado.



Fonte - Autoria própria.

Diferentemente das abordagens em que o *cliente* é embutido no *veículo* (como VO), aqui ele é tratado como entidade separada, e sua identificação é feita por meio de um campo simples (por exemplo, *id_cliente*), equivalente a uma chave estrangeira conceitual. Isso viabiliza a separação de responsabilidades e modelagem orientada à consulta direta do *veículo*, com a possibilidade de acessar o *cliente*, se necessário, em uma consulta complementar.

4.2.7.2. Aplicação no Apache Cassandra

Na sétima abordagem, a modelagem adota uma estrutura em que na tabela de *veículo*, o *cliente* é referenciado por meio de um identificador, sem estar embutido no registro. Nesse caso, o relacionamento é do tipo associação 1:1 por referência, no qual o campo *id_cliente* é armazenado diretamente na tabela *veículos*, funcionando como uma chave estrangeira conceitual. Essa modelagem permite que os dados de *veículo* e *cliente* sejam mantidos em tabelas separadas e independentes, evitando a duplicação de dados e favorecendo a separação de responsabilidades. A criação da tabela *veículos*, com referência ao *cliente*, é demonstrada na Figura 65, e a tabela *clientes* segue estrutura sem fazer referência a tabela *veículos*, como demonstrada na Figura 66.

Figura 65 - *Script* de CQL para criação da tabela *veículos* com identificador de *cliente*.

```
CREATE TABLE IF NOT EXISTS veiculos (
  placa text PRIMARY KEY,
  cor text,
  id_cliente int
);
```

Fonte - Autoria própria.

Figura 66 - *Script* de CQL para criação da tabela *clientes* sem elementos embutidos e referências.

```
CREATE TABLE IF NOT EXISTS clientes (
  id int PRIMARY KEY,
  nome text
);
```

Fonte - Autoria própria.

Essa estrutura é adequada para aplicações que priorizam o acesso direto ao *veículo* e que acessam os dados do *cliente* somente quando necessário. Quando for preciso obter os dados de ambos, é necessário realizar duas consultas separadas, conforme exemplificado na Figura 67.

Figura 67 - *Script* de CQL para consulta nas tabelas *veículos* e *clientes*.

```
token@cqlsh:experimento> SELECT * FROM veiculos WHERE placa = 'JKL1234';

 placa   | cor    | id_cliente
---------+--------+------------
 JKL1234 | Prata |          1

(1 rows)
token@cqlsh:experimento> SELECT * FROM clientes WHERE id = 1;

 id | nome
----+-----------------
  1 | Eduardo Martins

(1 rows)
```

Fonte - Autoria própria.

Assim como nas outras abordagens com associação por referência, o Cassandra não garante integridade entre as tabelas, cabendo à aplicação garantir que o *id_cliente* referenciado exista e esteja consistente. Essa abordagem é vantajosa em termos de baixo acoplamento e escalabilidade,

especialmente quando os dados do cliente não precisam ser carregados em todas as operações com veículo.

4.2.7.3. Avaliação da abordagem

Vantagens

- **Baixa redundância:** Os dados do *cliente* não são duplicados na tabela de *veículos* nem na tabela de *clientes*, o que economiza espaço e facilita a manutenção dos dados.
- **Maior independência entre os agregados:** *Veículo* e *cliente* são modelados como entidades completamente separadas, o que facilita o versionamento e a evolução de cada estrutura.
- **Simplicidade na atualização dos dados**: Como os dados do *cliente* e *veículo* estão em cada uma em sua tabela distinta, as atualizações afetam apenas um local, evitando a necessidade de sincronização com registros embutidos.
- **Maior escalabilidade:** As entidades *cliente* e *veículos* por estarem em agregados distintos cada um será particionado entre os nós do *cluster* de forma apartada, e incentivando a distribuição dos dados. Com isso, essa abordagem cria registros menores que conseguem ser facilmente escaláveis.

Desvantagens

- **Perda de atomicidade:** As alterações nos dados do *veículo* e do *cliente* são feitas em tabelas distintas, sem garantias transacionais entre elas.
- **Consulta composta exige múltiplas operações:** Para obter as informações completas do *veículo* e do *cliente*, é necessário realizar duas consultas, uma para cada entidade, o que aumenta a latência da operação.
- **Ausência de integridade referencial:** Como o Cassandra não oferece suporte a chaves estrangeiras, a existência do *id_cliente* em *veículos* não é verificada automaticamente, podendo gerar referências inválidas.
- **Maior complexidade em leituras agregadas:** Casos em que é necessário exibir informações completas (*veículo* + *cliente*) requerem lógica de junção na aplicação, o que pode dificultar a manutenção e a escalabilidade.

Consistência

A consistência é mantida apenas dentro de cada tabela. Como não há garantias transacionais entre *veículo* e *cliente*, a consistência entre os agregados é eventual e deve ser controlada pela aplicação.

Disponibilidade

As tabelas funcionam de maneira independente. A indisponibilidade de uma delas não compromete o funcionamento da outra, mas afeta operações que exigem dados completos.

Tolerância à partição

Essa abordagem apresenta alta tolerância à partição, pois cada um dos agregados são distribuídos de forma independente. Além de manter o tamanho dos registros de acordo com o recomendado pelo Apache Cassandra.

Desempenho

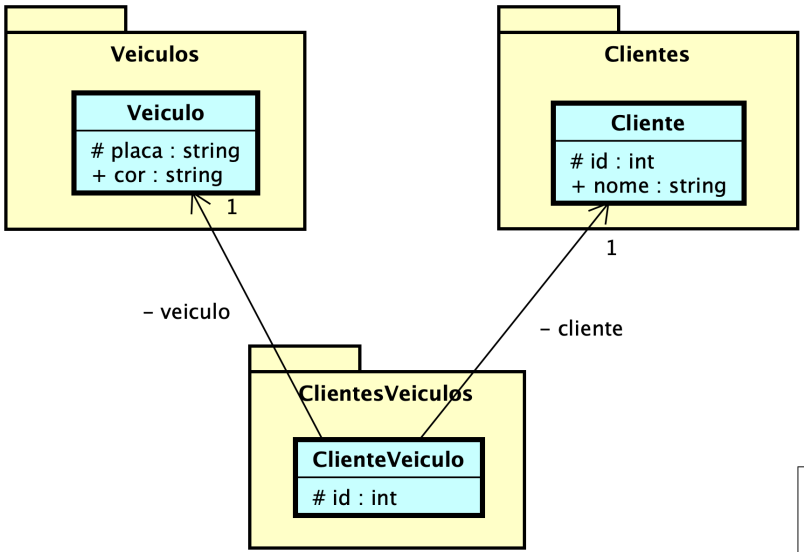
A leitura de dados do *veículo* e *cliente* é rápida e eficiente, pois os dados são acessados diretamente pela chave primária. No entanto, operações que requerem também os dados (*veículo* + *cliente*) são mais custosas, exigindo operações adicionais. Por outro lado, a escrita é leve e localizada, e o custo de armazenamento é menor devido à ausência de redundância.

4.2.8. Tabela intermediária de referências

4.2.8.1. Modelagem lógica em AML

Na oitava e última abordagem analisada, representada na Figura 68, adota-se uma tabela intermediária para gerenciar o relacionamento entre *cliente* e *veículo*. Nesse modelo, tanto *cliente* quanto *veículo* são modelados como agregados independentes, armazenados em suas respectivas tabelas, e uma terceira tabela é criada para manter as referências cruzadas entre eles. Essa tabela intermediária funciona como uma tabela de junção, permitindo identificar quais *veículos* pertencem a um determinado cliente e vice-versa, sem embutir nem duplicar dados entre as entidades principais. Essa estrutura é útil em contextos onde a frequência de leitura e atualização entre as entidades é equilibrada, ou quando se deseja manter agregados totalmente isolados, evitando acoplamento e duplicidade, mas ainda assim permitir associação eficiente entre eles. Outro cenário útil é quando é necessário manter um relacionamento N:M.

Figura 68 - Modelagem lógica da abordagem da tabela intermediária de referências.



Fonte - Autoria própria.

4.2.8.2. Aplicação no Apache Cassandra

Na oitava abordagem, a modelagem utiliza uma tabela intermediária de referências para representar o relacionamento entre as entidades *cliente* e *veículo*, de forma a manter ambos os agregados completamente desacoplados. Nessa estrutura, *cliente* e *veículo* são armazenados em tabelas separadas, cada um como raiz do próprio agregado, e uma terceira tabela é criada exclusivamente para manter a associação entre eles, representando o relacionamento "1:N" ou "N:M", conforme a necessidade da aplicação. No contexto do Apache Cassandra, isso é implementado com uma tabela intermediária que armazena os campos *id_cliente* e *placa*, que formam a chave primária composta da tabela, como ilustrado na Figura 62.

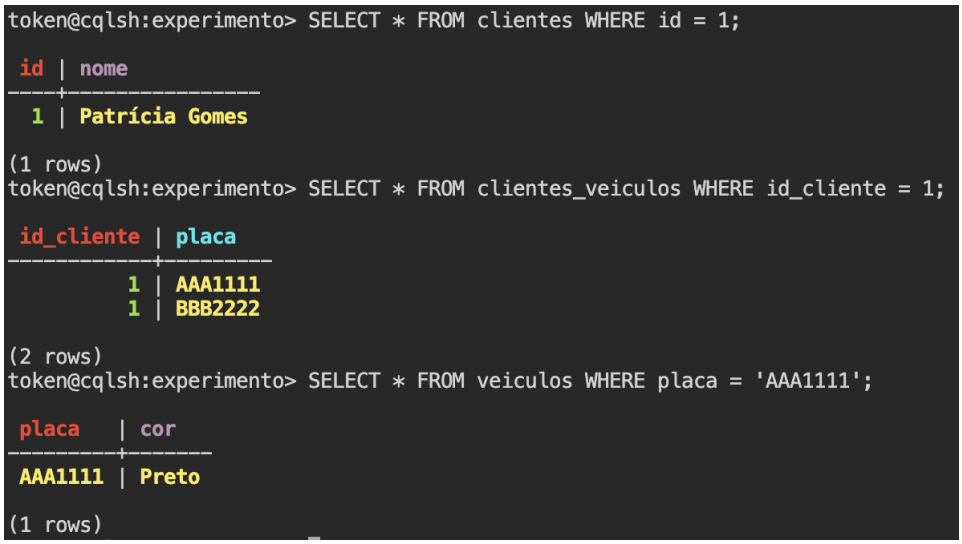
Figura 69 - *Script* de CQL para criação da tabela *intermediária*.

```
CREATE TABLE IF NOT EXISTS clientes_veiculos (
  id_cliente int,
  placa text,
  PRIMARY KEY (id_cliente, placa)
);
```

Fonte - Autoria própria.

Essa abordagem oferece a maior flexibilidade de consulta e expansão, permitindo, por exemplo, evoluir o relacionamento para N:N se necessário. No entanto, requer que os dados completos de *cliente* e *veículo* sejam consultados separadamente nas suas respectivas tabelas após recuperar os vínculos da tabela *intermediária*, como demonstrado na Figura 70.

Figura 70 - *Script* de CQL para consulta nas tabelas *veículos*, *intermediária* e *clientes*.



Fonte - Autoria própria.

Assim como nas demais abordagens com referência, a consistência entre as tabelas deve ser garantida pela aplicação, que precisa controlar a integridade das associações na tabela intermediária. Essa modelagem é indicada para cenários complexos, onde o relacionamento entre entidades é dinâmico, e há necessidade de independência total entre os agregados.

4.2.8.3. Avaliação da abordagem

Vantagens

- **Separação total de responsabilidades entre os agregados:** *Cliente* e *veículo* são tratados como entidades completamente independentes, o que facilita a manutenção, versionamento e escalabilidade de cada um.

- **Ausência de redundância nos dados principais:** Nenhuma informação é duplicada entre as tabelas de *clientes* e *veículos*, resultando em menor uso de armazenamento e menos risco de inconsistência.
- **Flexibilidade para relacionamento N:N no futuro**: A estrutura pode ser adaptada para representar relações N:N, se necessário, com a simples inclusão de mais registros na tabela intermediária.
- **Consultas mais organizadas para grandes volumes**: Ideal para cenários onde os volumes de dados são altos e os relacionamentos são dinâmicos, evitando a manutenção de listas dentro de agregados.

Desvantagens

- **Maior complexidade de leitura do agregado completo:** leitura do conjunto de *veículos* de um *cliente* exige: (1) consultar a tabela intermediária para obter as *placas*, (2) realizar múltiplas leituras individuais na tabela *veículos*, o que aumenta o número de operações e a latência.
- **Ausência de integridade referencial:** Como Cassandra não possui chaves estrangeiras, a aplicação precisa garantir que as *placas* listadas na tabela *intermediária* existam na tabela *veiculos*.
- **Ausência de atomicidade em operações cruzadas**: A consistência entre *clientes_veiculos*, *clientes* e *veículos* deve ser controlada manualmente pela aplicação. Não há transações entre essas tabelas.
- **Custo adicional de manutenção da tabela *intermediária*:** Toda vez que um relacionamento é criado, atualizado ou removido, a tabela *intermediária* deve ser atualizada, adicionando uma etapa adicional em todas as operações de escrita relacionadas ao vínculo.

Consistência

Cada tabela mantém consistência apenas em seu próprio escopo. A consistência entre os relacionamentos armazenados na tabela intermediária e os dados reais das entidades referenciadas é eventual e exige controle manual pela aplicação.

Disponibilidade

As três tabelas operam de forma independente. A falha de qualquer uma afeta apenas as funcionalidades específicas associadas a ela. No entanto, a leitura completa do relacionamento entre *cliente* e seus *veículos* exige que todas as tabelas estejam disponíveis.

Tolerância à partição

A abordagem mantém alta tolerância à partição, pois cada entidade será escalado de forma distinta, com isso diminuindo o tamanho do registro, e facilitando a distribuição do dado no *cluster*.

Desempenho

A leitura dos dados principais de *cliente* ou *veículo* é eficiente, pois cada entidade é acessada diretamente pela chave primária em sua respectiva tabela. No entanto, a leitura do agregado completo - ou seja, obter todos os *veículos* de um *cliente* - é mais custosa, pois requer múltiplas etapas: primeiro, consultar a tabela *intermediária* para recuperar os identificadores dos *veículos*; em seguida, realizar leituras individuais na tabela *veículos* para obter os dados completos de cada um. Esse processo pode aumentar a latência e gerar *overhead*, especialmente em cenários com muitos relacionamentos. Por outro lado, a escrita de dados é leve para operações simples em *clientes* ou *veículos*, mas torna-se mais complexa ao exigir atualizações adicionais na tabela *intermediária* para manter a referência entre as entidades. Quanto ao armazenamento, a abordagem é eficiente, pois não há duplicação de dados - apenas armazenamento de chaves de referência na tabela de relacionamento.

5. RESULTADOS

Os resultados obtidos neste experimento prático demonstram que a AML é capaz de representar com clareza os principais conceitos, características e restrições do Apache Cassandra, sendo compatível com sua estrutura de armazenamento de dados orientados à família de colunas. A AML mostrou-se eficiente na modelagem de aspectos fundamentais, como:

- Armazenamento desnormalizado de estruturas agregadas (dados embutidos), reforçando o conceito de agregados do DDD;
- Atomicidade por linha, garantindo consistência transacional dentro de um mesmo agregado;
- Definição de chaves primárias, com uso adequado de *partition key* e *clustering key*;
- Uso de coleções nativas como *list*, *set* e *map*, respeitando suas restrições específicas, como:
    - *set*: garante unicidade dos elementos;
    - *map*: exige unicidade das chaves;
    - Coleções de UDTs devem ser frozen para que possam ser utilizadas;
- Criação de UDTs para representar objetos de valor e estruturas complexas;
- Utilização de colunas *STATIC*, para armazenar atributos compartilhados por todas as linhas de uma mesma partição;
- Modelagem orientada à leitura, adaptando a estrutura dos agregados para otimizar diferentes padrões de acesso.

Os construtores da AML - como composição, composição qualificada, associação, atributos com multiplicidade, elementos embutidos, referenciados e redundantes - foram aplicados com sucesso no Cassandra, demonstrando que a linguagem possui expressividade suficiente para mapear as decisões de modelagem típicas de BDs *NoSQL* orientados à família de colunas.

Para auxiliar nas decisões de modelagem lógica em BDs *NoSQL*, cuja complexidade aumenta devido ao suporte ao armazenamento de estruturas agregadas, este trabalho analisou as dimensões de consistência, disponibilidade, tolerância a partições, desempenho e escalabilidade em oito abordagens distintas de modelagem lógica, todas aplicadas sobre um mesmo modelo conceitual. Com isso, os experimentos demonstraram que estruturas desnormalizadas, como nas abordagens de número 1 e 4, favorecem consistência transacional, atomicidade e menor latência nas leituras de agregados completos, pois os dados são recuperados com uma única consulta. No entanto, essas

abordagens apresentaram limitações de flexibilidade para consultas específicas (por exemplo, por atributos do VO) e maior risco de crescimento do registro além do permitido, o que compromete a escalabilidade horizontal do Cassandra. Já as abordagens com tabelas redundantes (como nas abordagens 2, 3 e 5) permitiram consultas mais flexíveis, adaptadas a diferentes padrões de acesso, mas com um custo: maior complexidade nas operações de escrita e riscos de inconsistência entre as estruturas. Como o Cassandra não oferece suporte transacional entre tabelas, a aplicação precisa controlar isso, por meio de *BEGIN BATCH* ou verificações manuais de consistência. Por outro lado, as abordagens com elementos referenciados (como nas abordagens 6 e 7) favorecem separação de responsabilidades, baixo acoplamento, maior escalabilidade e uso eficiente de armazenamento. No entanto, exigiram consultas adicionais para compor os dados agregados, aumentando a latência. E por fim, a abordagem 8, com tabela intermediária de referência, destacou-se pela sua flexibilidade estrutural e possibilidade de representar relações “N:N”. Porém, exigiu lógica adicional na aplicação para manter a integridade dos relacionamentos, pois o controle de consistência entre as três tabelas depende do sistema externo ao banco. De forma geral, os resultados confirmam que não existe uma abordagem única ideal: cada estrutura apresenta *trade-offs*, e a escolha deve ser feita com base nas regras de negócio, nos padrões de acesso da aplicação e nas restrições técnicas do Apache Cassandra. A Tabela 5 resume os principais resultados comparativos entre as abordagens.

Tabela 5 – Resumo dos resultados das abordagens de modelagem do experimento

Abordagem	**Principais Vantagens**	**Principais Desvantagens**
Tabela com *array* de elementos embutidos	Alta *performance* de leitura e consistência local	Baixa flexibilidade na consulta dos VOs
Tabela com *array* de elementos embutidos e tabela redundante	Otimização na leitura	Risco de inconsistência entre os elementos
Tabela com *array* de elementos embutidos e tabela redundante com apenas um elemento embutido	Consulta eficiente e completa a partir de ambos os agregados	Redundância completa, alto risco de inconsistência e alto custo de sincronização dos dados
Tabela com apenas um elemento embutido	Leitura e simplificada por Veículo	Cliente duplicado em múltiplos registros. Em caso de alteração, necessidade em alterar vários registros
Tabela com apenas um elemento embutido e tabela redundante	Otimização na leitura	Risco de inconsistência entre os elementos
Tabela com *array* de elementos referenciados	Separação de responsabilidades, maior escalabilidade e menor redundância	Maior latência em consulta de agregados

Abordagem	**Principais Vantagens**	**Principais Desvantagens**
Tabela com um elemento referenciado	Modelo simples, maior escalabilidade e menor redundância	Maior latência em consulta de agregados principalmente para consulta de informações que envolve todos os registros de cliente
Tabela intermediária de referências	Alta flexibilidade nas consultas e possibilidade de relacionamento N:N	Necessidade de controle total por parte da aplicação

6. CONCLUSÃO

Este trabalho explorou, de forma prática, a aplicação da linguagem AML na modelagem lógica de BD *NoSQL* orientados à família de colunas, utilizando como base o Apache Cassandra. A partir de oito abordagens distintas, foi possível validar que a AML é uma linguagem expressiva, funcional e compatível com os princípios de agregados do DDD, sendo adequada para representar estruturas complexas e adaptáveis às características do Cassandra.

Ao longo do experimento, constatou-se que a AML oferece suporte suficiente para mapear tanto as restrições do Cassandra - como atomicidade por linha, exigência de frozen em coleções de UDTs, e uso de chaves de partição e ordenação - quanto seus recursos nativos de modelagem, como *collections*, UDTs, colunas *STATIC* e estruturas desnormalizadas. A análise das abordagens permitiu entender melhor os impactos das decisões de modelagem sobre os requisitos não funcionais, como consistência, escalabilidade e desempenho. Mais do que apontar a melhor estrutura, o trabalho reforça que a escolha da modelagem lógica deve ser orientada pelas necessidades do domínio de negócio, padrões de acesso da aplicação e limitações da tecnologia utilizada. A AML se mostra promissora como ferramenta de apoio nesse processo, promovendo clareza na definição de agregados e maior controle sobre o desenho das estruturas de dados.

Como trabalhos futuros, sugere-se aplicar a AML em outros modelos de bancos *NoSQL*, como os orientados a chave-valor; explorar experimentos com modelos conceituais mais complexos; desenvolver ferramentas que automatizam a transformação de diagramas AML em scripts CQL; e realizar *benchmarks* quantitativos para comparar o desempenho entre as abordagens propostas.

7. REFERÊNCIAS

[1] LÓSCIO, Bernadette Farias; OLIVEIRA, HR de; PONTES, JC de S. NoSQL no desenvolvimento de aplicações Web colaborativas. VIII Simpósio Brasileiro de Sistemas Colaborativos, v. 10, n. 1, p. 11, 2011. Disponível em: <https://www.researchgate.net/profile/Bernadette-Loscio/publication/268201466_NoSQL_no_desenvolvimento_de_aplicacoes_Web_colaborativas/links/576aa72008aef2a864d1ef8c/NoSQL-no-desenvolvimento-de-aplicacoes-Web-colaborativas.pdf>. Acesso em: 4 jan. 2025.

[2] LIMA, C. Projeto Lógico de Bancos de Dados NoSQL Documento a Partir de Esquemas Conceituais Entidade-Relacionamento Estendido (EER). 2016. Tese de Doutorado. Master's thesis, PPGCC-UFSC. Disponível em: <https://repositorio.ufsc.br/handle/123456789/167633>. Acesso em: 4 jan. 2025.

[3] SOUSA, Flávio RC et al. Gerenciamento de dados em nuvem: Conceitos, sistemas e desafios. Topicos em sistemas colaborativos, interativos, multimidia, web e bancos de dados, Sociedade Brasileira de Computacao, p. 101-130, 2010. Disponível em: <https://d1wqtxts1xzle7.cloudfront.net/4660346/gerenciamento_dados_nuvem-libre.pdf?1390837848=&response-content-disposition=inline%3B+filename%3DGerenciamento_de_Dados_em_Nuvem_Conceito.pdf&Expires=1739667966&Signature=Zn8NvzQ9IxT7u3i4LvU2kCgOdGqW3F-f39qgeEpC4FgrzMDm8mCYzFHZlOxzszv670zX-SZfpt4SKDQQ0cTkTywNru2EOZz3nhGjreU8GiSWobsEy825mFqbPu3nlE4Xsb1aiQPv9A13PVbcNTP3x4fHooA91v2U~9RyYHsR2qCidy9PRIPaviSxWtGl~NkJcZrxjhwPnWKq02bm2Z8HwFTQfoNBsFHhgenaen91wB2DwDTr~uuPTNsDjkB0qJMfOn58gjXfFOYKMm5hReBd-nOH3eOjviuNN0PtCXT9AUUb5xeJ5B~msojYPOVzuFrnm7nTwRWMgRa9qqPzLRdAeQ__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA>. Acesso em 04 jan. 2025.

[5] DB-ENGINES RANKING. Popularity Ranking of Database Management Systems. Disponível em: <https://db-engines.com/en/ranking>. Acesso em: 4 jan. 2025.

[6] SADALAGE, Pramod J.; FOWLER, Martin. NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education, 2013. Acesso em 04 jan. 2025.

[7] CHEBOTKO, Artem; KASHLEV, Andrey; LU, Shiyong. A big data modeling methodology for Apache Cassandra. In: 2015 IEEE International Congress on Big Data. IEEE, 2015. p. 238-245. Disponível em:

<https://ieeexplore.ieee.org/abstract/document/7207225>. Acesso em 04 jan. 2025.

[8] POFFO, João Paulo. Projeto lógico de bancos de dados NoSQL colunares a partir de esquemas conceituais entidade-relacionamento estendido (EER). 2016. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2016. Disponível em: <https://repositorio.ufsc.br/xmlui/handle/123456789/167985>. Acesso em 04 jan. 2025.

[9] MONTEIRO, Thomas Anderson Feitosa. Modelagem de banco de dados orientados a documentos com AML. 2023. Trabalho de Conclusão de Curso. Disponível em: <https://repositorio.ufpe.br/handle/123456789/52945>. Acesso em 04 jan. 2025.

[10] BREWER, Eric. CAP twelve years later: How the" rules" have changed. Computer, v. 45, n. 2, p. 23-29, 2012. Disponível em: <https://ieeexplore.ieee.org/abstract/document/6133253>. Acesso em 04 jan. 2025.

[11] ANURADHA, J. et al. A brief introduction on Big Data 5Vs characteristics and Hadoop technology. Procedia computer science, v. 48, p. 319-324, 2015. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050915006973>. Acesso em 04 jan. 2025.

[12] ALBUQUERQUE, Henrique Mauricio de. NoSQL: características e aplicações. 2017. Disponível em: <https://ric.cps.sp.gov.br/handle/123456789/690>. Acesso em em 04 jan. 2025.

[13] STRAUCH, Christof; KRIHA, Walter. NoSQL databases. Lecture Notes, Stuttgart Media University, v. 20, n. 24, p. 79, 2011. Disponível em: <https://www.researchgate.net/profile/Jesus-Sanchez-Cuadrado/publication/257491810_A_repository_for_scalable_model_management/links/568baf0508ae051f9afc5857/A-repository-for-scalable-model-management.pdf>. Acesso em em 04 jan. 2025.

[14] APACHE CASSANDRA. Cassandra Documentation. Disponível em: <https://cassandra.apache.org/doc/latest/>. Acesso em em 04 jan. 2025.

[15] DATASTAX. DataStax Academy. Disponível em: <https://www.datastax.com/academy>. Acesso em

em 04 jan. 2025.

[16] APACHE CASSANDRA. Cassandra Documentation. Disponível em: <https://cassandra.apache.org/doc/latest/>. Acesso em em 04 jan. 2025.

[27] APACHE CASSANDRA. Cassandra Basics. Disponível em: <https://cassandra.apache.org/_/cassandra-basics.html>. Acesso em em 04 jan. 2025.

[18] DATASTAX. Cassandra OSS Documentation: Write Path. Disponível em: <https://docs.datastax.com/en/cassandra-oss/2.1/cassandra/dml/dml_write_path_c.html>. Acesso em em 04 jan. 2025.

[19] DATASTAX. Cassandra OSS Documentation: About Reads. Disponível em: <https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlAboutReads.html>. Acesso em em 04 jan. 2025.

[20] HEUSER, Carlos Alberto. Projeto de banco de dados: Volume 4 da Série Livros didáticos informática UFRGS. Bookman Editora, 2009.

[21] BATINI, Carlo; CERI, Stefano; NAVATHE, Shamkant B. Conceptual database design: an entity-relationship approach. Benjamin-Cummings Publishing Co., Inc., 1991.

[22] EVANS, Eric. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

[23] VERNON, Vaughn. Domain-driven design distilled. Addison-Wesley Professional, 2016.

[24] DATASTAX. What is Cassandra?. Disponível em: <https://www.datastax.com/guides/what-is-cassandra>. Acesso em em 04 jan. 2025.

[25] WIKIPEDIA. Apache Cassandra. Disponível em: <https://en.wikipedia.org/wiki/Apache_Cassandra>. Acesso em: Acesso em em 04 jan. 2025.

[26] BENTLEY, Jon. Programming pearls: The back of the envelope. Communications of the ACM, v. 27,

n. 3, p. 180-184, 1984. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/357994.381168>. Acesso em em 04 jan. 2025.

[27] REINSEL, David; GANTZ, John; RYDNING, John. Data Age 2025: The Evolution of Data Growth, Use, and the Emerging Edge. An IDC White Paper, patrocinado por Seagate, abr. 2017. Disponível em: <https://www.seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf>. Acesso em em 04 jan. 2025.

[28] DOMO. Data Never Sleeps 11. Disponível em: <https://www.domo.com/learn/infographic/data-never-sleeps-11>. Acesso em em 04 jan. 2025.