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

LUIZ PHILLIP PEREIRA BARBOSA

MODELAGEM DE BANCOS DE DADOS DE FAMÍLIA DE COLUNAS COM AML:
Uma Análise Comparativa no Apache HBase

Recife
2025

LUIZ PHILLIP PEREIRA BARBOSA

MODELAGEM DE BANCOS DE DADOS DE FAMÍLIA DE COLUNAS COM AML:
Uma Análise Comparativa no Apache HBase

Trabalho de Conclusão de Curso apresentado ao Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para a obtenção do título de bacharel em Sistemas de Informação.

Orientador (a): Robson do 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

Barbosa, Luiz Phillip Pereira.
MODELAGEM DE BANCOS DE DADOS DE FAMÍLIA DE COLUNAS COM AML: Uma Análise Comparativa no Apache HBase / Luiz Phillip Pereira Barbosa. - Recife, 2025.
86 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. Aggregate Modeling Language. 2. Apache HBase. 3. Banco de Dados NoSQL. 4. Banco de Dados de Família de Colunas. 5. Modelagem Lógica. 6. Domain-Driven Design. I. Fidalgo, Robson do Nascimento. (Orientação). II. Lima, Gênesis Jeferson Ferreira Pereira de. (Coorientação). IV. Título.

000 CDD (22.ed.)

LUIZ PHILLIP PEREIRA BARBOSA

MODELAGEM DE BANCOS DE DADOS DE FAMÍLIA DE COLUNAS COM AML:
Uma Análise Comparativa no Apache HBase

Trabalho de Conclusão de Curso apresentado ao Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para a obtenção do título de bacharel em Sistemas de Informação.

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

A modelagem de bancos de dados NoSQL apresenta desafios distintos dos modelos relacionais tradicionais, exigindo abordagens que priorizem eficiência no acesso aos dados, escalabilidade e consistência eventual. Este trabalho investiga a aplicação da *Aggregate Modeling Language* (AML) na modelagem de bancos de família de colunas, utilizando o HBase como plataforma. Fundamentada nos conceitos de agregados do *Domain-Driven Design* (DDD), a AML propõe uma estruturação lógica que pode otimizar operações de leitura e escrita, reduzindo a fragmentação e melhorando a organização dos dados. A pesquisa realiza uma análise comparativa de diferentes estratégias de modelagem, explorando cenários que variam desde a incorporação total dos dados dentro de agregados até abordagens baseadas em referências. São discutidos os impactos dessas estratégias em termos de desempenho, redundância, flexibilidade de consulta e complexidade de manutenção. A partir dessa análise, busca-se fornecer diretrizes para o uso da AML em bancos de família de colunas, contribuindo para a tomada de decisão na modelagem de dados altamente distribuídos e escaláveis.

Palavras-chave: Aggregate Modeling Language (AML), Apache HBase, Banco de Dados NoSQL, Banco de Dados de Família de Colunas, Modelagem Lógica, Domain-Driven Design (DDD).

ABSTRACT

The modeling of NoSQL databases presents challenges that differ from traditional relational models, requiring approaches that prioritize data access efficiency, scalability, and eventual consistency. This study investigates the application of the *Aggregate Modeling Language* (AML) in modeling column-family databases, using HBase as a platform. Based on the concepts of aggregates from *Domain-Driven Design* (DDD), AML proposes a logical structure that can optimize read and write operations, reducing fragmentation and improving data organization. The research conducts a comparative analysis of different modeling strategies, exploring scenarios ranging from fully embedded data within aggregates to reference-based approaches. The impacts of these strategies are discussed in terms of performance, redundancy, query flexibility, and maintenance complexity. From this analysis, the study aims to provide guidelines for using AML in column-family databases, contributing to decision-making in the modeling of highly distributed and scalable data systems.

Keywords: Aggregate Modeling Language (AML), Apache HBase, NoSQL Database, Column-Family Database, Logical Modeling, Domain-Driven Design (DDD).

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AML *Aggregate Modeling Language*

CAP *Consistência, Disponibilidade e Particionamento*

CP *Consistência e Particionamento*

DDD *Domain-Driven Design*

DER *Diagrama de Entidade e Relacionamento*

DSML *Domain-Specific Modeling Language*

EER *Enhanced Entity-Relationship*

HBase *Hadoop Database (Apache HBase)*

JSON *JavaScript Object Notation*

NoSQL *Not Only SQL*

SGBD *Sistema Gerenciador de Banco de Dados*

UML *Unified Modeling Language*

VO *Value Object*

WAL *Write-Ahead Logging*

SUMÁRIO

1. INTRODUÇÃO

1.1. CONTEXTO

A crescente digitalização de processos em diversas áreas tem gerado um volume exponencial de dados, frequentemente não estruturados e distribuídos em larga escala. Esse crescimento impõe desafios significativos aos bancos de dados relacionais, cujos modelos tradicionais de armazenamento e recuperação de dados nem sempre conseguem oferecer a flexibilidade, escalabilidade horizontal e alta disponibilidade demandadas por aplicações modernas. Nesse contexto, os bancos de dados NoSQL emergiram como uma alternativa viável, permitindo um gerenciamento mais eficiente de grandes volumes de dados distribuídos. Entre esses bancos, o HBase se destaca como uma solução de família de colunas, amplamente utilizada em sistemas distribuídos que exigem desempenho otimizado e armazenamento escalável.

Embora os bancos NoSQL ofereçam vantagens significativas em termos de escalabilidade e desempenho, a modelagem lógica dos dados ainda representa um desafio, pois não segue as diretrizes rígidas dos bancos relacionais. Decisões como quando embutir dados em um mesmo registro ou distribuí-los em coleções distintas afetam diretamente a eficiência das operações de leitura e escrita, bem como a consistência dos dados. Nesse contexto, a *Aggregate Modeling Language* (AML) surge como uma abordagem promissora para a modelagem de bancos NoSQL, incluindo bancos de família de colunas como o HBase. Baseada nos conceitos de agregados do *Domain-Driven Design* (DDD), a AML proporciona um modelo estruturado e intuitivo para representar a organização dos dados, permitindo uma modelagem mais alinhada às necessidades de escalabilidade e desempenho dessas bases. Sua simplicidade e aderência ao paradigma NoSQL tornam-na uma ferramenta valiosa, especialmente em contextos que exigem eficiência na leitura e escrita.

Neste trabalho, foi analisada a aplicação da AML na modelagem de bancos de família de colunas, utilizando o HBase como plataforma de estudo. Através da investigação de diferentes estratégias de modelagem, buscou-se demonstrar a

capacidade da AML em representar de forma completa os elementos e estruturas típicas do HBase, estabelecendo um mapeamento direto entre seus conceitos e a organização lógica desse tipo de banco.

1.2. MOTIVAÇÃO E JUSTIFICATIVA

A modelagem lógica de bancos NoSQL de família de colunas é ainda um tema em constante evolução, de modo que a escolha das melhores estratégias pode se mostrar desafiadora. Embora a literatura já consagre boas práticas de modelagem muito bem estabelecidas e aceitas para bancos relacionais, não se observa um consenso igualmente consolidado no caso de bancos colunares, como o HBase. Diante desse cenário, a AML se apresenta como uma alternativa para estruturar a modelagem lógica, mas sua aplicação prática em bancos de família de colunas ainda é pouco explorada. Assim, torna-se necessário investigar sua completude, limitações e impactos na organização dos dados, avaliando oportunidades de aprimoramento e sua viabilidade como ferramenta para otimizar esquemas de armazenamento em bancos de dados colunares.

A relevância desse estudo se dá pela necessidade de fornecer um referencial técnico para a aplicação da AML no contexto de bancos NoSQL colunares, demonstrando não apenas sua utilidade prática, mas também sua completude como linguagem de modelagem lógica compatível com o Apache HBase. Ao analisar diferentes estratégias de modelagem e seus impactos no desempenho e na consistência dos dados, o trabalho busca consolidar a AML como uma alternativa viável e robusta para ambientes altamente distribuídos.

1.3. OBJETIVOS

1.3.1. Objetivo Geral

Este trabalho tem como objetivo investigar e demonstrar a aplicação da AML na modelagem lógica de bancos de dados de família de colunas, utilizando o Apache HBase como plataforma. Além disso, visa evidenciar a completude da AML

ao representar, de forma sistemática, os principais conceitos do modelo colunar do HBase.

1.3.2. Objetivos Específicos

- Identificar os fundamentos teóricos dos bancos de dados NoSQL, com foco nos bancos de família de colunas e nas características do Apache HBase.
- Explorar as propriedades da *Aggregate Modeling Language* e sua relação com o *Domain-Driven Design*, com ênfase na modelagem lógica de agregados.
- Aplicar a AML na modelagem lógica de bancos de família de colunas, utilizando o HBase como plataforma para análise.
- Demonstrar como a Aggregate Modeling Language pode mapear com precisão os principais elementos do modelo de dados do Apache HBase, estabelecendo uma correspondência lógica entre os componentes da linguagem e a estrutura física do banco.

1.4. METODOLOGIA DE PESQUISA

A metodologia adotada nesta pesquisa combina revisão teórica e desenvolvimento empírico, visando investigar e demonstrar a aplicação prática da *Aggregate Modeling Language* em bancos de dados de família de colunas, com ênfase no Apache HBase. Para estruturar o estudo, a abordagem foi dividida em três passos macro:

- **Fundamentação Teórica:** Nesta etapa, realiza-se uma revisão bibliográfica e documental para estabelecer as bases conceituais do trabalho, abordando os fundamentos do *Domain-Driven Design*, o modelo de agregados, as particularidades dos bancos NoSQL e modelagem lógica. Essa fase permite mapear os elementos essenciais que orientarão a modelagem dos dados.
- **Aggregate Modeling Language (AML):** Com base na revisão teórica, esta etapa foca na análise e definição dos conceitos e notações da AML. São detalhados os elementos estruturais, tais como entidades, objetos de valor,

atributos, links e pictogramas, e sua relação com os conceitos do Apache HBase.

- **Aplicação e Avaliação de Cenários de Modelagem:** Com o embasamento teórico e a definição da AML, são desenvolvidos múltiplos cenários práticos de modelagem para o Apache HBase. Cada cenário representa uma estratégia diferente de organização dos dados, como uso de embutimento, referências ou redundância. O objetivo central desta etapa é demonstrar, na prática, a completude da AML ao cobrir, de forma sistemática e expressiva, as diferentes possibilidades de modelagem lógica compatíveis com a estrutura do HBase. As abordagens são avaliadas segundo critérios de desempenho, redundância, complexidade de manutenção e implicações no modelo CAP. Os resultados são então consolidados em diretrizes para a escolha da estratégia mais adequada a cada contexto.

2. FUNDAMENTAÇÃO TEÓRICA

A fim de estabelecer um embasamento teórico sólido para este trabalho, esta seção explora os principais conceitos relacionados à modelagem de bancos de dados NoSQL de família de colunas, com foco na aplicação da AML. Inicialmente, serão apresentadas as principais diferenças entre o modelo relacional e os bancos NoSQL, destacando as limitações dos bancos de dados relacionais e os benefícios oferecidos pelas abordagens não relacionais, especialmente no contexto de dados distribuídos e escaláveis. Em seguida, será discutido o funcionamento dos bancos NoSQL, abordando suas categorias e modelos de armazenamento, com ênfase no modelo orientado a colunas, representado pelo Apache HBase.

Na segunda parte do capítulo, será explorada a modelagem de bancos de dados de família de colunas, detalhando os desafios desse paradigma e a necessidade de abordagens especializadas. Nesse contexto, será introduzida a AML, explicando sua origem e sua relação com o DDD. Por fim, será discutida a representação diagramática da AML, incluindo seus pictogramas, nós e links, demonstrando como esses elementos podem ser utilizados para estruturar modelos lógicos de forma eficiente.

2.1. MODELO RELACIONAL vs NÃO-RELACIONAL

2.1.1. Limitações do Modelo Relacional

O modelo relacional apresenta desafios significativos no que tange à escalabilidade horizontal, uma vez que foi concebido para operar de maneira centralizada ou, no máximo, com replicação limitada. Para distribuir os dados entre múltiplos servidores, técnicas como *sharding* devem ser empregadas, o que acarreta aumento da complexidade operacional, especialmente na manutenção da consistência e integridade dos dados (VENKATRAMAN, 2016). Além da escalabilidade, a rigidez do esquema constitui outra limitação relevante. A estrutura tabular impõe a necessidade de um esquema predefinido, com colunas fixas e tipos de dados bem definidos, o que pode se tornar um entrave em cenários que exigem alterações frequentes na estrutura da aplicação. Segundo Da Silva Dias (2023)

modelos mais flexíveis, como os bancos orientados a documentos ou chave-valor, permitem uma adaptação mais dinâmica às mudanças nos requisitos dos sistemas. O custo computacional de consultas complexas também representa um fator crítico. Operações que envolvem múltiplas tabelas e *joins* tendem a comprometer o desempenho do banco de dados relacional à medida que o volume de dados cresce, tornando-se um desafio para aplicações que demandam performance.

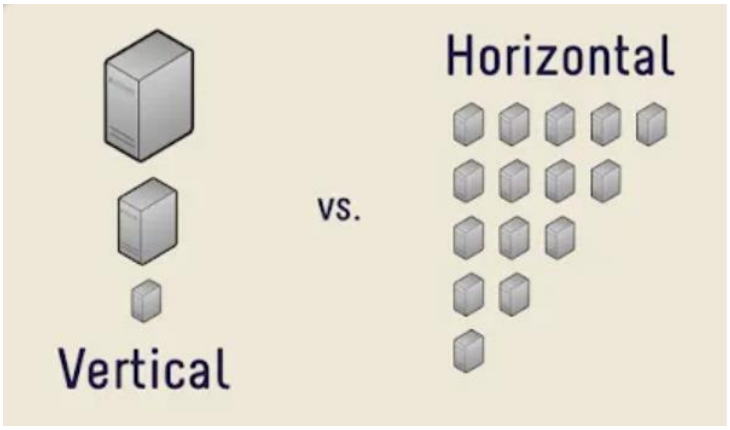
Diante dessas limitações, embora os bancos de dados relacionais permaneçam amplamente utilizados, a necessidade de maior escalabilidade e flexibilidade impulsionou a adoção de bancos NoSQL, que oferecem arquiteturas mais adequadas a ambientes distribuídos e de grande volume de dados (GARCIA, 2019).

2.1.2. Bancos NoSQL

Os bancos de dados NoSQL surgiram como uma resposta às limitações dos bancos relacionais. A necessidade de armazenar e processar grandes volumes de informações de forma distribuída impulsionou o desenvolvimento de novas abordagens, sendo que os primeiros modelos amplamente reconhecidos incluem o *BigTable*, do Google, e o *DynamoDB*, da *Amazon*. O termo NoSQL consolidou-se a partir de 2009, quando Johan Oskarsson organizou um evento que destacou projetos como Cassandra, HBase, MongoDB e CouchDB, estabelecendo uma nova categoria de sistemas de gerenciamento de dados (SHAHIDA, 2022).

Esses bancos de dados apresentam características distintas que os diferenciam dos modelos relacionais. Conforme Schreiner et al. (2015), destacam-se a escalabilidade horizontal, ilustrada na figura 1, a distribuição dos dados em múltiplos servidores, a flexibilidade na definição de esquemas e a alta disponibilidade. A estrutura não tabular permite que diferentes tipos de dados sejam armazenados sem a necessidade de um esquema rígido, tornando-os mais adaptáveis a aplicações dinâmicas.

Figura 1 – Representação de escalabilidade horizontal e vertical



Fonte: HADDADI (2020)

Uma das mudanças mais evidentes com o NoSQL é o afastamento do modelo relacional. Cada solução NoSQL utiliza um modelo de dados diferente, que pode ser classificado em quatro categorias amplamente adotadas no ecossistema NoSQL: chave-valor, documentos, família de colunas e grafos. Desses, os três primeiros compartilham uma característica comum em seus modelos de dados, chamada de de orientação a agregados, conceito essencial para modelagens mais eficientes e flexíveis em bancos de dados distribuídos (EVANS, 2004).

2.1.2.1. Modelos de Dados Orientados a Agregados

O modelo relacional organiza os dados dividindo-os em tuplas (ou linhas de uma tabela). Essa abordagem, embora eficiente em muitos casos, apresenta limitações, pois não permite a inclusão de estruturas mais complexas, como registros aninhados ou listas de valores dentro de uma única tupla. Essa simplicidade é, contudo, um dos pilares do modelo relacional, garantindo operações consistentes e previsíveis ao manipular conjuntos de dados (KAUFMANN; MEIER, 2023).

A orientação a agregados adota um conceito diferente, reconhecendo que, em muitas situações, os dados são melhor representados em unidades estruturadas mais ricas do que meros conjuntos de tuplas (SADALAGE; FOWLER, 2013). Nesse contexto, pode ser mais vantajoso modelar informações como registros compostos, permitindo o aninhamento de listas e subestruturas. Essa abordagem é amplamente utilizada em bancos de dados chave-valor, orientados a documentos e de famílias

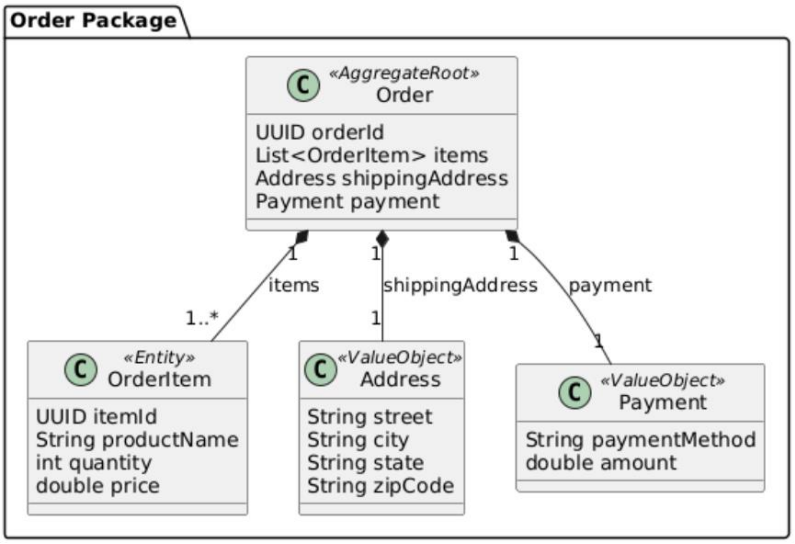
de colunas, que lidam com dados organizados dessa forma. Essa estrutura é chamada de agregado.

Segundo Evans (2004), o conceito de agregado tem suas raízes no DDD, em que um agregado representa um conjunto de objetos interconectados que devem ser tratados como uma única unidade lógica. Essa estrutura é usada tanto para manipulação de dados quanto para garantir a consistência das operações. Uma característica fundamental dos agregados é a atualização atômica, ou seja, todas as mudanças realizadas dentro de um agregado são tratadas como uma única operação indivisível. A forma como os bancos de dados chave-valor, orientados a documentos e família de colunas operam está em sintonia com esse conceito. Trabalhar com agregados facilita a distribuição dos dados em clusters, pois cada agregado pode ser facilmente replicado ou particionado (*sharding*) de maneira independente (SADALAGE; FOWLER, 2013). Além disso, essa organização costuma ser mais intuitiva para desenvolvedores, pois muitos sistemas já interagem com dados agrupados dessa forma (SCHREINER et al., 2020).

O modelo de agregados é composto por três elementos principais: *Aggregate Root* (Raiz do Agregado), *Entity* (Entidade) e *Value Object* (Objeto de Valor), cada um desempenhando um papel bem definido na organização dos dados. Dentro de um agregado, existe uma entidade principal denominada *Aggregate Root*, que governa todas as interações dentro da estrutura e impede que entidades internas sejam acessadas ou modificadas diretamente. Isso significa que todas as alterações nos dados contidos dentro de um agregado devem ocorrer por meio da *Aggregate Root*, garantindo que as regras de negócio sejam aplicadas corretamente e evitando inconsistências (EVANS, 2004). Além da *Aggregate Root*, há Entidades, que são objetos identificáveis e passíveis de modificação ao longo do tempo. Diferentemente dos objetos de valor, as entidades possuem identidade única, geralmente representada por um identificador (ID). Mesmo que seus atributos sejam alterados, a identidade permanece a mesma. Em contrapartida, os *Value Objects* não possuem identidade própria e geralmente são imutáveis, sendo usados para representar conceitos que podem ser copiados e compartilhados entre diferentes entidades dentro do agregado. Esses objetos são frequentemente utilizados para modelar atributos descritivos, como endereços ou características técnicas de um item (SADALAGE; FOWLER, 2013).

A figura 2 ilustra um cenário de um sistema de e-commerce onde tem-se que um Pedido (***Order***) pode ser considerado um *Aggregate Root*, pois contém diversas entidades associadas, como itens do pedido, informações de pagamento e endereço de entrega.

Figura 2 – Agragado Order e Listagem de Código do Agregado Order

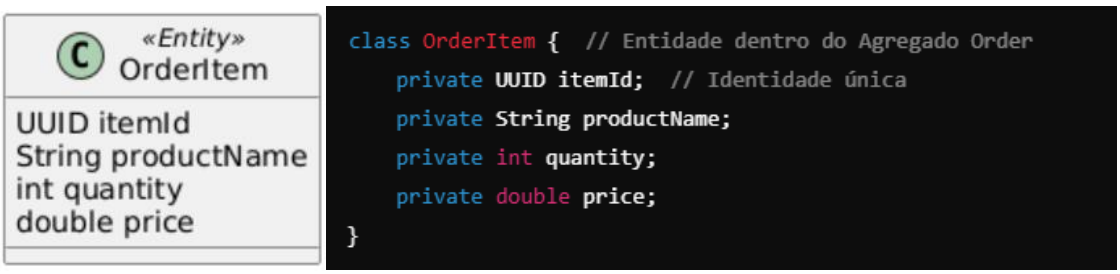



```java
class Order {  // Aggregate Root
    private UUID orderId;
    private List<OrderItem> items;  // Entidade
    private Address shippingAddress;  // Objeto de Valor
    private Payment payment;  // Objeto de Valor
}
```

Fonte: Autoria Própria (2025)

A figura 3 apresenta o contexto de um Pedido, onde um Item do Pedido (***OrderItem***) pode ser uma Entidade, pois ele tem um identificador único dentro do agregado e pode ser atualizado ou removido. Mesmo que o preço do item ou a quantidade sejam alterados, o item ainda será a mesma entidade porque seu ***itemId*** permanece o mesmo.

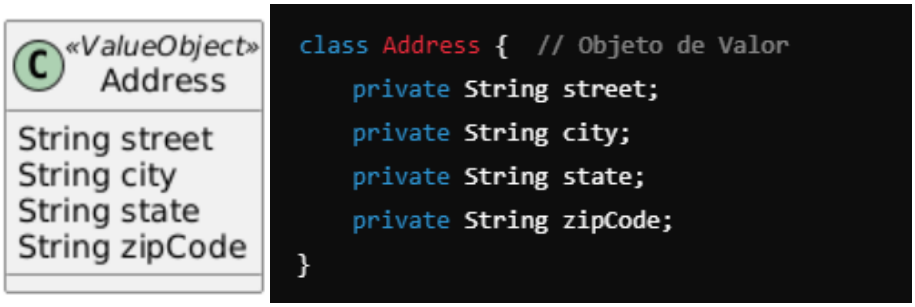
Figura 3 – Entidade OrderItem e Listagem de Código da Entidade OrderItem



Fonte: Autoria Própria (2025)

Um Endereço de Entrega (***Shipping Address***) pode ser um Objeto de Valor dentro do agregado Pedido (figura 4), pois ele não precisa ter um identificador único. Se dois pedidos tiverem o mesmo endereço, eles podem compartilhar a mesma instância do objeto de valor.

Figura 4 – Objeto de Valor Adress e Listagem de Código do Objeto de Valor Adress

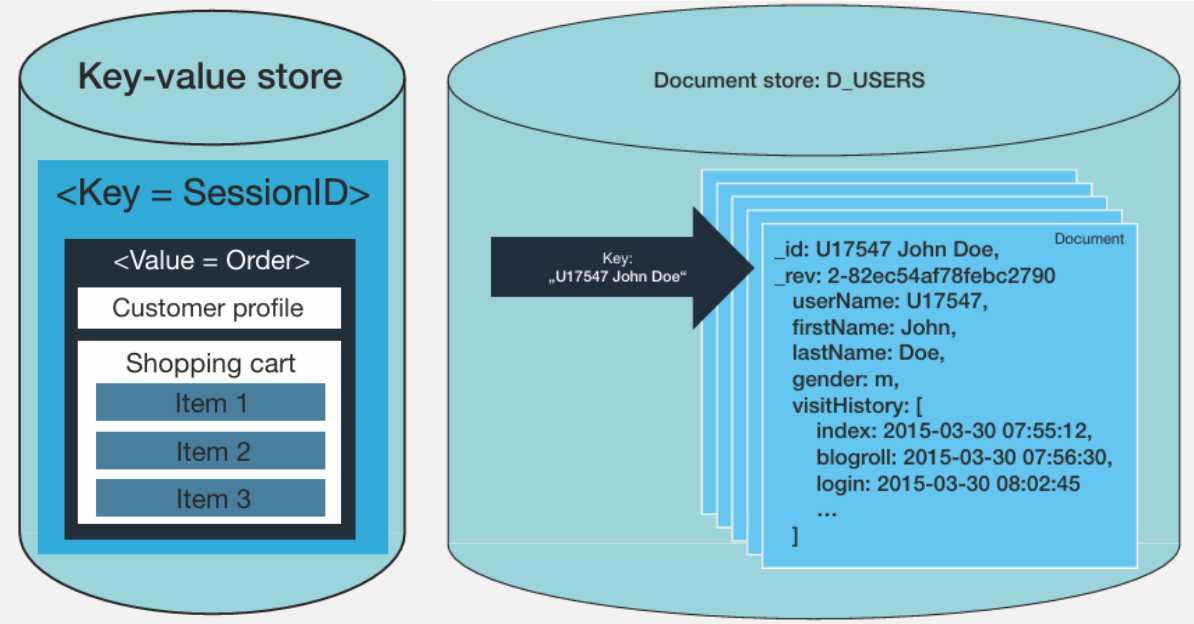


Fonte: Autoria Própria (2025)

2.1.2.1.1. Modelos Chave-Valor e Orientados a Documentos

Os bancos de dados chave-valor e orientados a documentos, respectivamente ilustrados na figura 5, são amplamente reconhecidos como exemplos de modelos orientados a agregados, pois organizam seus dados em unidades autônomas, acessadas por meio de um identificador único (ADYA et al., 2019). Em ambos os casos, a informação é armazenada em agregados, que podem ser recuperados a partir de uma chave associada, oferecendo alto desempenho e escalabilidade em cenários distribuídos (MONTEIRO, 2023).

Figura 5 – Representação do modelo de dados chave-valor e orientado a documentos



Fonte: KAUFFMAN (2023)

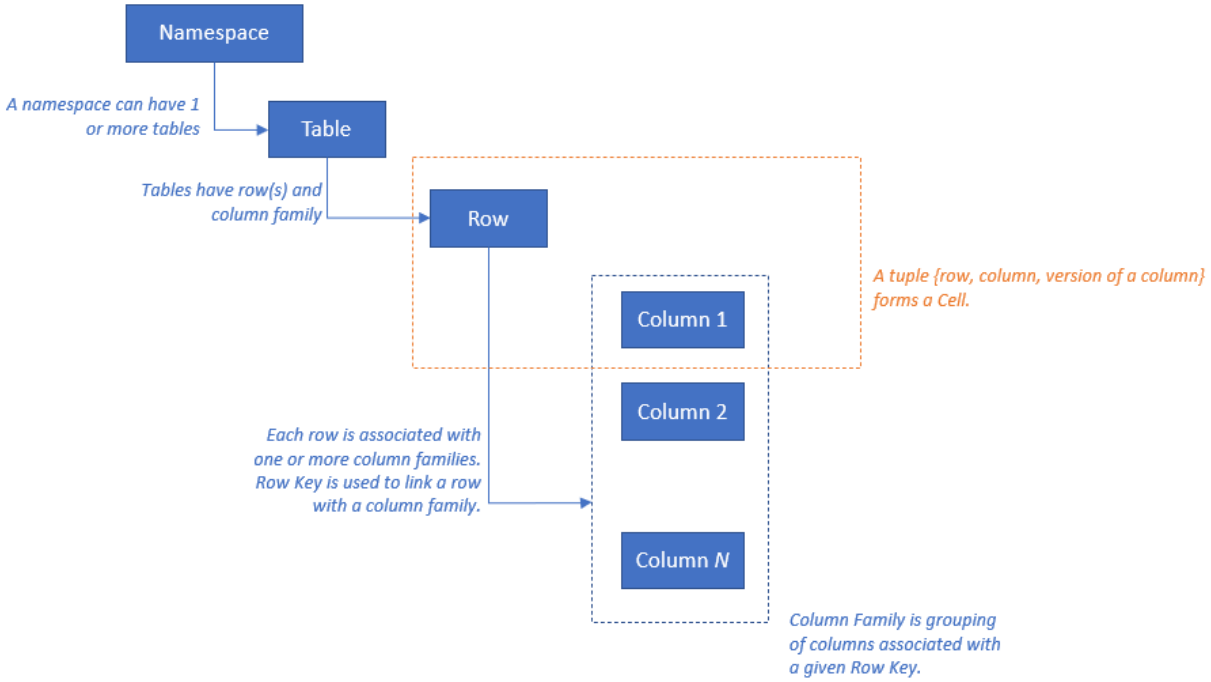
A principal distinção entre esses dois modelos está na forma como o banco de dados interpreta a estrutura do agregado. Segundo Shahida (2022), nos bancos de dados chave-valor, os agregados são armazenados de maneira opaca, ou seja, o sistema trata cada agregado como um bloco de dados indivisível, sem interpretar sua composição interna. Em contrapartida, os bancos orientados a documentos mantêm uma estrutura interna bem definida, permitindo ao sistema compreender e manipular os componentes individuais do agregado.

2.1.2.1.2. Modelo Orientados a Colunas

Como mencionado anteriormente, um dos primeiros bancos de dados NoSQL com estrutura orientada a colunas foi o *BigTable*, desenvolvido pelo Google. Seu nome remete a uma organização tabular, implementada por meio de colunas esparsas e sem a necessidade de um esquema fixo (SOARES; BOSCARIOLI, 2013). Essa estrutura não deve ser entendida rigidamente como uma tabela tradicional, mas sim como um mapa de dois níveis, em que cada “linha” corresponde a um agregado e suas respectivas colunas são armazenadas de forma flexível (figura 6). Essa abordagem influenciou significativamente sistemas subsequentes,

como HBase e Cassandra, que aprimoraram o conceito de *column family* em ambientes distribuídos (TSAI et al., 2022; SHETH, 2023).

Figura 6 – Representação do modelo de dados colunar



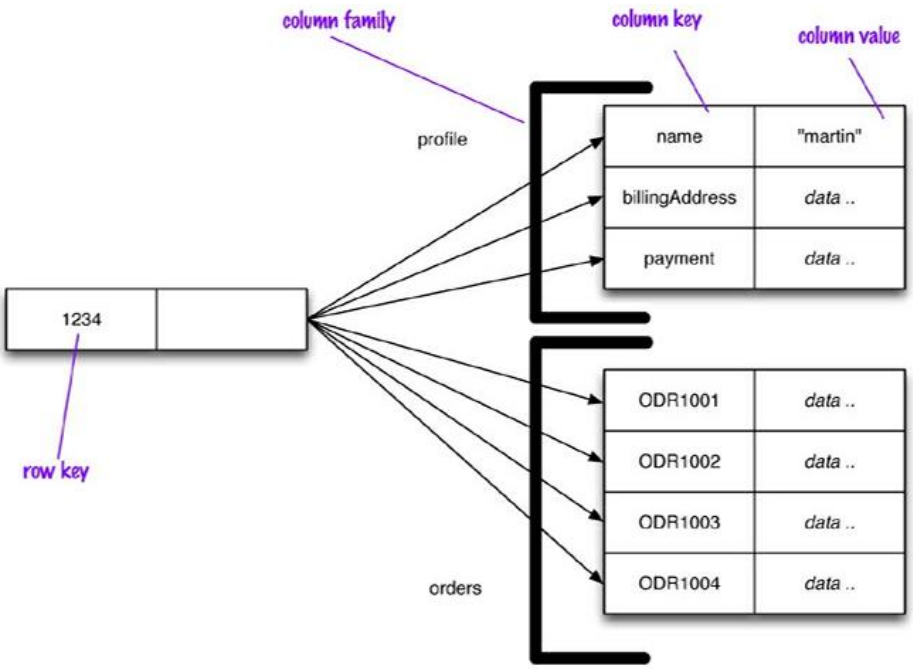
Fonte: MICROSOFT LEARN (2024)

Em certa medida, bancos de dados colunarmente orientados guardam similaridades com bancos relacionais, mas a principal diferença reside na forma de armazenamento. Enquanto os relacionais utilizam linhas como unidade fundamental, o que facilita a gravação sequencial, algumas aplicações se beneficiam mais de leituras em colunas específicas ao longo de diversas linhas. Para esses cenários, o armazenamento baseado em grupos de colunas torna-se mais eficiente, justificando o nome de bancos orientados a colunas (FROZZA, 2022).

O modelo de famílias de colunas pode ser entendido como uma estrutura agregada em dois níveis. Tal qual ocorre nos bancos chave-valor, a chave principal geralmente é vista como um identificador de linha, que representa um agregado. A distinção está em tratar esse agregado como um mapa de valores detalhados, em que os elementos secundários são as colunas (figura 7). Assim, é possível acessar toda a linha ou apenas colunas específicas; por exemplo, para recuperar o nome de um cliente, seria utilizada uma operação do tipo ***get('1234', 'name')*** (SADALAGE; FOWLER, 2013). Além do *get*, bancos de dados colunares disponibilizam a operação de varredura (*scan*), que permite percorrer intervalos de linhas ou colunas

de modo sequencial (HADDADI, 2020). Esse recurso viabiliza a leitura contínua de grandes conjuntos de dados, fundamental em cenários de análise intensiva, onde se deseja agrupar ou filtrar informações distribuídas em diferentes regiões do cluster.

Figura 7 – Representação do modelo de dados colunar



Fonte: SADALAGE and FOWLER (2013)

Os bancos de famílias de colunas organizam suas colunas dentro de famílias de colunas, sendo que cada coluna deve pertencer a uma única família. Como consequência, assume-se que os dados pertencentes a uma mesma família de colunas serão acessados conjuntamente.

Essa estrutura possibilita diferentes formas de organização dos dados:

- **Orientação por linhas:** Cada linha representa um agregado (por exemplo, um cliente identificado por um ID), e as famílias de colunas organizam segmentos úteis desse agregado, como perfil do cliente e histórico de pedidos.
- **Orientação por colunas:** Cada família de colunas define um tipo de registro (por exemplo, perfis de clientes), onde cada linha corresponde a um registro distinto. Dessa forma, uma linha pode ser interpretada como a junção de diferentes registros armazenados nas diversas famílias de colunas.

Em bancos de dados de famílias de colunas, cada linha não está restrita a um conjunto fixo de colunas. Isso significa que colunas adicionais podem ser incluídas em linhas específicas conforme necessário, permitindo que linhas diferentes tenham conjuntos de colunas completamente distintos. Durante o uso normal do banco de dados, novas colunas podem ser adicionadas dinamicamente a uma linha, mas a criação de novas famílias de colunas é uma operação menos frequente e pode até exigir a interrupção do banco de dados (SADALAGE; FOWLER, 2013). A figura 7 ilustra outro aspecto dos bancos de famílias de colunas, que pode parecer incomum para usuários acostumados com bancos de dados relacionais: a modelagem da família de colunas de pedidos (*orders*). Como colunas podem ser adicionadas livremente, listas de itens podem ser modeladas representando cada item como uma coluna separada. Esse conceito pode parecer estranho se o banco de dados for interpretado como uma tabela relacional, mas se torna intuitivo quando se compreende que uma linha em um banco de famílias de colunas é, na verdade, um agregado.

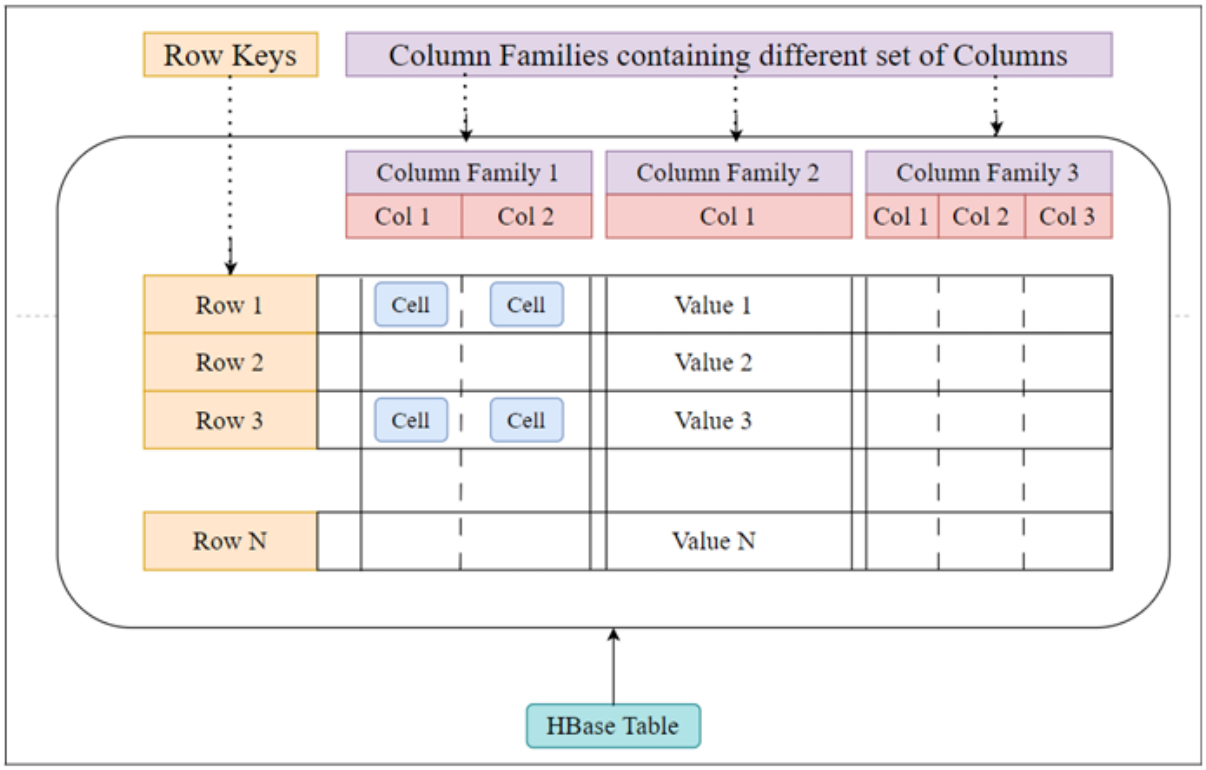
2.1.2.1.2.1. Apache HBase

O HBase é um sistema de gerenciamento de banco de dados NoSQL *open source*, desenvolvido para operar sobre a infraestrutura do Apache Hadoop. Segundo Haddadi, Oulahyane e Oufttou (2020), sua concepção foi inspirada no Google *BigTable*, visando ao armazenamento e processamento distribuído de grandes volumes de dados estruturados e semiestruturados. A arquitetura do HBase possibilita a replicação automática de dados e o particionamento horizontal entre diversos nós do cluster, garantindo escalabilidade e resiliência (TSAI et al., 2022). Suas principais características incluem alta taxa de escrita, baixa latência e um design voltado para tolerância a falhas. Como banco de dados orientado a colunas, sua organização baseia-se na estruturação e recuperação de dados por colunas, em vez de linhas, o que otimiza o armazenamento e melhora a eficiência em operações que envolvem grandes quantidades de informação (HASSAN et al., 2021).

O HBase adota um modelo de armazenamento baseado em colunas e estruturado como um mapa hierárquico de chave-valor (figura 8). Sua principal vantagem é a flexibilidade, permitindo a adição ou remoção dinâmica de colunas

sem comprometer o desempenho. Por armazenar dados em formato de bytes, o HBase não impõe restrições quanto a tipos de dados específicos, possibilitando o processamento de informações semiestruturadas (SHETH, 2023).

Figura 8 – Modelo de Dados do HBase



Fonte: SHETH (2023)

Os principais componentes do modelo de dados do HBase são:

- **Tabela**: Em HBase, as tabelas possuem uma estrutura lógica, e não física. Elas são compostas por múltiplas linhas, distribuídas em regiões com base no intervalo da chave da linha (*rowkey*).
- **Linha**: As linhas representam uma abstração lógica, sendo formadas por combinações de famílias de colunas. O identificador único da linha, conhecido como *rowkey*, funciona como um índice primário.
- **Família de Colunas**: Os dados são organizados em famílias de colunas, que agrupam colunas relacionadas. As famílias são definidas no esquema da tabela e compartilhadas entre todas as linhas da tabela, mas cada linha pode conter um conjunto distinto de colunas dentro de cada família.

- **Coluna**: Definida dentro de uma família de colunas, uma coluna é composta por um nome, um valor e um *timestamp*. Para acessar uma coluna específica, utiliza-se um qualificador no formato *familia_coluna:nome_coluna.*

- **Célula**: O menor nível de armazenamento do HBase, uma célula contém o valor de um dado. Cada célula é identificada pelo triplo (*rowkey*, coluna, versão), sendo armazenada como um *array* de *bytes* (*byte*[]), caso a versão não seja informada é assumido que é a versão mais recente.

- **Versão**: Como cada célula pode conter múltiplas versões dos dados, os valores são armazenados em ordem decrescente de *timestamp*, garantindo que a versão mais recente seja acessada primeiro.

De acordo com o teorema CAP (Consistência, Disponibilidade e Particionamento), o HBase é classificado como um sistema CP, o que significa que, em situações de falha na comunicação da rede, ele prioriza a consistência em detrimento da disponibilidade. Essa escolha garante que todas as réplicas de dados permaneçam sincronizadas, mesmo em cenários de falhas ou particionamento de rede (TSAI et al., 2022).

O modelo de consistência do HBase apresenta as seguintes características (HADDADI, 2020):

1. **Consistência Forte em Nível de Linha**: O HBase garante que operações de leitura e escrita em uma mesma linha sejam fortemente consistentes. Isso significa que qualquer modificação realizada em uma linha será refletida em leituras subsequentes imediatamente após a operação ser concluída.

2. **Replicação Consistente**: Para assegurar durabilidade e tolerância a falhas, o HBase mantém múltiplas réplicas dos dados em diferentes nós do cluster. Embora a replicação ocorra de maneira assíncrona, o sistema garante que todas as cópias eventualmente sejam sincronizadas. Mecanismos como *Write-Ahead Logging* (WAL) e *distributed commit logs* são empregados para garantir que as réplicas se mantenham consistentes.

A arquitetura do HBase permite lidar com falhas de rede sem comprometer a integridade dos dados. Assim que a conectividade é restabelecida, o sistema recupera automaticamente os dados e assegura a consistência entre as réplicas.

Essa robustez torna o HBase adequado para aplicações que exigem confiabilidade e integridade dos dados, como sistemas financeiros e plataformas de e-commerce. No entanto, o modelo de consistência forte pode introduzir alguma latência, especialmente em comparação com sistemas que priorizam disponibilidade sobre consistência.

2.2. MODELAGEM DE BANCOS DE DADOS

A modelagem de bancos de dados é um processo fundamental para garantir a eficiência, escalabilidade e confiabilidade de um sistema de armazenamento de dados. Em projetos de menor porte, a implementação pode ocorrer diretamente na camada física; contudo, em sistemas mais complexos, um processo estruturado de modelagem é essencial para atender às necessidades informacionais dos usuários e assegurar requisitos críticos, como disponibilidade e desempenho (DORNELLES, 2025).

Esse processo é geralmente composto por três fases principais. No modelo conceitual, representa-se o domínio do negócio por meio de entidades, atributos e relacionamentos de forma independente de qualquer tecnologia. De acordo com Al-Ghifari e Azizah (2022), o Diagrama de Entidade e Relacionamento (DER) é um dos principais artefatos dessa etapa, pois facilita a comunicação entre a equipe técnica e os usuários. Em seguida, o modelo lógico traduz esses conceitos para uma estrutura compatível com um Sistema Gerenciador de Banco de Dados (SGBD), considerando aspectos técnicos e estruturais, mas ainda sem vinculação a um SGBD específico. Em bancos relacionais, essa fase envolve a definição de tabelas, chaves e regras de integridade referencial. Já em bancos NoSQL, os dados podem ser organizados em coleções, famílias de colunas ou grafos, conforme as particularidades do modelo de dados adotado (POFFO, 2016).

Por fim, o modelo físico implementa a estrutura delineada no modelo lógico em um SGBD específico, incluindo definições sobre organização de dados em disco, indexação, particionamento e segurança (MA et al., 2024). Em bancos relacionais, isso se traduz na criação de tabelas e definição dos tipos de dados. Nos bancos NoSQL, adapta-se ao modelo de armazenamento adotado, como documentos BSON no *MongoDB* ou famílias de colunas no Apache HBase (LIMA, 2015). Um

planejamento físico adequado assegura que o sistema não apenas atenda às necessidades funcionais do negócio, mas também cumpra requisitos de escalabilidade e eficiência de acesso aos dados.

2.2.1. Modelagem Lógica de Bancos Orientados a Colunas

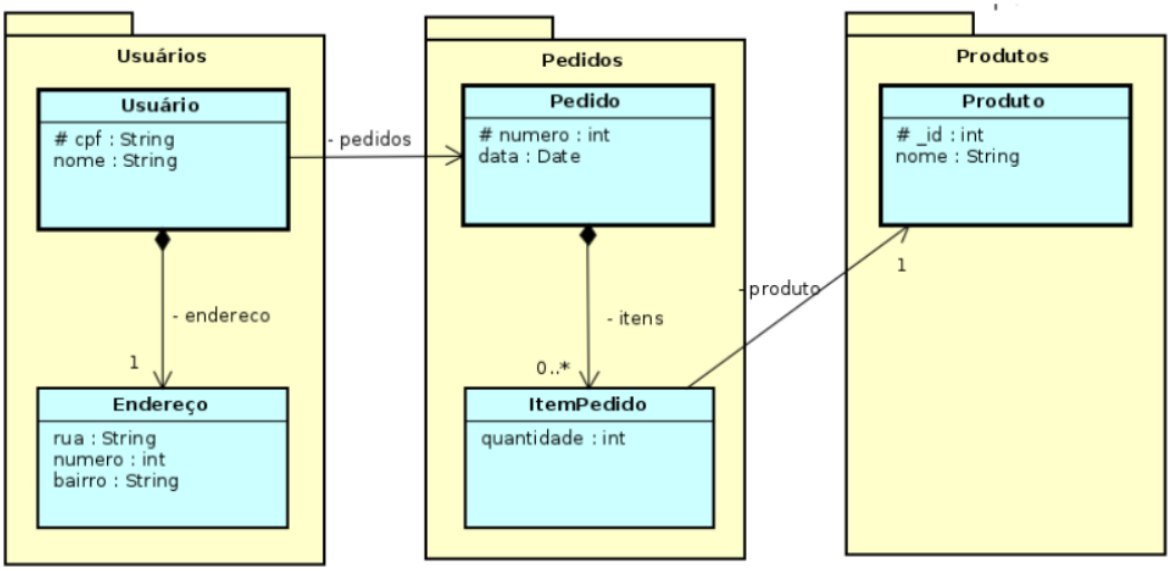
A modelagem lógica de bancos de dados NoSQL impõe desafios devido à ausência de um esquema rígido e à diversidade de paradigmas, conforme discutido na seção anterior. Para transformar um esquema conceitual EER em um modelo NoSQL sem a necessidade de regras de conversão excessivamente específicas, adota-se um modelo intermediário capaz de capturar características comuns entre diferentes abordagens (FROZZA, 2022).

Neste trabalho, será utilizada a *Aggregate Modelling Language* (AML) como linguagem de modelagem lógica para um banco de dados colunar. A escolha da AML justifica-se por ela ser uma *Domain-Specific Modeling Language* (DSML), ou seja, uma linguagem projetada especificamente para a modelagem de dados orientados a agregados. Por ser classificada com uma DSML, a AML possui uma sintaxe concreta (com notação visual bem definida), uma sintaxe abstrata (representada por um metamodelo), e uma semântica estática rigorosa, que define regras de boa formação (LIMA, 2018). Isso significa que a linguagem impede que o projetista construa modelos estruturalmente incorretos, garantindo maior confiabilidade no projeto. No contexto da AML, essa validação garante, por exemplo, que toda entidade (Entity) possua um identificador; que não seja possível conectar duas entidades diretamente sem o uso de um link de associação; que cada entidade seja a raiz de um agregado; e que um agregado contenha apenas uma única entidade raiz. Da mesma forma, os objetos de valor (*Value Objects* – VOs) devem sempre estar conectados a uma entidade, obrigatoriamente por meio de um link de composição, não podendo existir de forma isolada no modelo. Essa validação semântica automática permite que erros estruturais sejam detectados ainda na fase de modelagem, antes da geração de código ou da transição para o modelo físico. Em analogia com linguagens de programação, a semântica estática da AML age como um compilador, bloqueando modelagens inválidas.

3. AGGREGATE MODELLING LANGUAGE - AML

A AML é uma linguagem de modelagem lógica voltada para bancos de dados NoSQL, desenvolvida para estruturar e organizar dados de forma eficiente em sistemas que adotam o modelo de agregados do DDD. Sua principal proposta é permitir a representação clara de entidades, objetos de valor e seus relacionamentos, otimizando a modelagem de domínios complexos e garantindo maior consistência e escalabilidade para sistemas distribuídos. Inspirada em elementos visuais da *Unified Modeling Language* (UML), ela facilita sua adoção por profissionais que já utilizam esta linguagem, além de ser compatível com ferramentas amplamente utilizadas no mercado. Entretanto, a AML se diferencia por possuir uma quantidade reduzida de construtores, cada um com semântica única e bem definida, tornando o processo de modelagem mais simples e direto (figura 9) (MONTEIRO, 2023).

Figura 9 – Modelagem de um Exemplo de E-Commerce



Fonte: MONTEIRO (2023)

3.1. REPRESENTAÇÃO DIAGRAMÁTICA DA AML

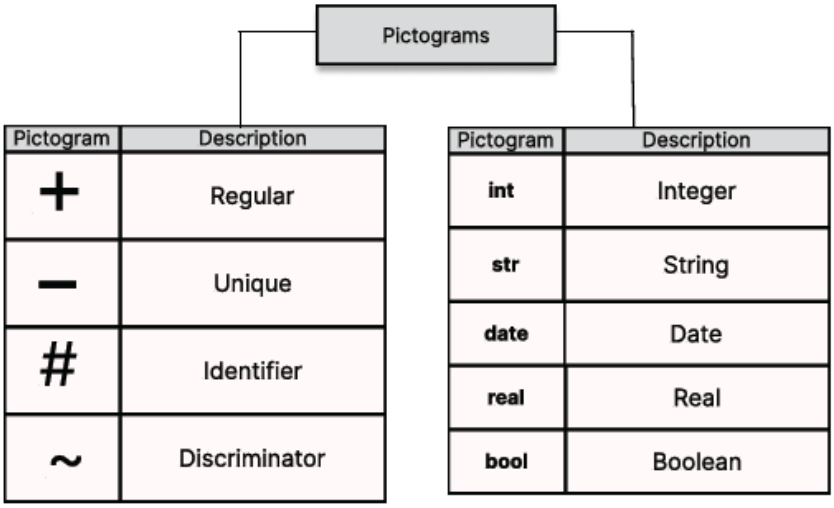
Para organizar a modelagem de forma clara e eficiente, a AML estrutura seus diagramas por meio de três tipos principais de construtores: nós, links e pictogramas. Os nós representam os principais elementos do modelo, incluindo atributos,

entidades e objetos de valor, compondo a base dos agregados. Os links estabelecem as conexões entre esses elementos, definindo os relacionamentos dentro do agregado e entre os agregados, garantindo a integridade da estrutura. Já os pictogramas atribuem semântica aos componentes modelados, proporcionando uma representação mais intuitiva e facilitando a interpretação do diagrama. Esses três construtores combinados permitem que a AML expresse de maneira visual e direta a organização dos dados em um banco NoSQL baseado em agregados.

3.1.1. Pictogramas

Na *Aggregate Modeling Language* (AML), os pictogramas são utilizados para fornecer informações adicionais sobre os atributos, entidades e relacionamentos no modelo lógico. Esses pictogramas ajudam a definir propriedades importantes, como unicidade e função de um campo dentro de um agregado, como visto na figura 10.

Figura 10 – Pictogramas da AML



Fonte: Documentação da AML

Os pictogramas modificadores indicam restrições e características dos atributos dentro de entidades e objetos de valor. São eles:

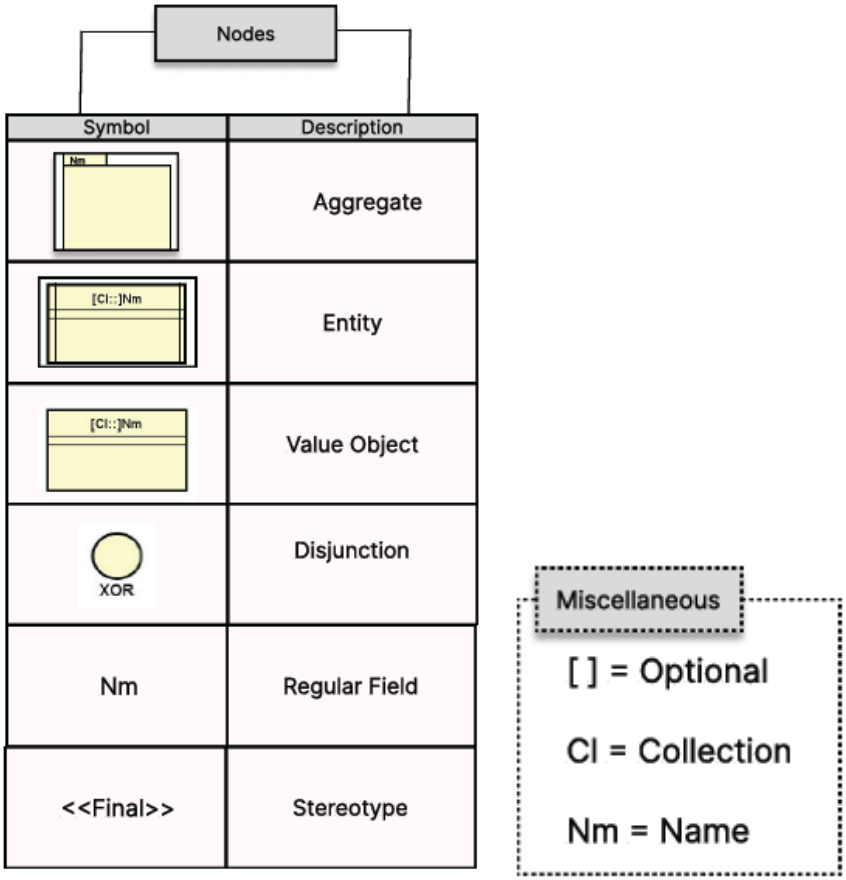
- **+ (*Regular*):** Indica que o atributo é acessível normalmente e pode ser referenciado sem restrições especiais. Este modificador acaba sendo opcional, visto que caso não possua uma das outras três indicações, o atributo é considerado regular.

- **- (*Unique*):** Define que o valor do atributo deve ser único dentro do contexto da entidade. Esse modificador impede a repetição de valores no mesmo conjunto de dados.
- # **(*Identifier*):** Marca um atributo como chave primária, ou seja, um identificador único que diferencia uma entidade de outra.
- **~ (*Discriminator*):** Representa um discriminador que auxilia na caracterização da unicidade de cada instância em dois cenários principais: primeiro, quando não existe um identificador natural (por exemplo, um *Value Object* sem atributo naturalmente único) e, segundo, quando a chave é composta (como em um banco colunar, onde a chave primária resulta da junção de uma *partition key* (pictograma “-”) e uma *clustering key* (pictograma “#”)). Em síntese, o discriminador atua de forma semelhante ao modelo EER, possibilitando distinguir instâncias que não possam ser identificadas por um único atributo ou que requeiram uma estrutura de identificação composta.

3.1.2. Nós e Seus Pictogramas

Através dos nós da AML é possível representar os seguintes conceitos: Atributo, entidade, objeto de valor e agregado. A figura 11 ilustra os tipos de nós que são utilizados para representar esses conceitos.

Figura 11 – Nós da AML



Fonte: Documentação da AML

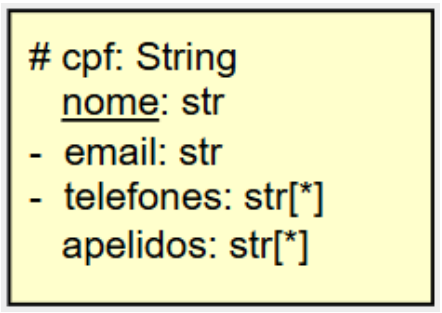
O primeiro conceito que é preciso definir é o de atributo. Na AML, um atributo representa uma característica ou propriedade de uma entidade ou objeto de valor dentro da modelagem de dados. Cada atributo armazena um valor específico e pode possuir modificadores que definem suas restrições e comportamento dentro do modelo. Os atributos podem ter diferentes tipos de dados, como inteiros, *strings*, datas, números reais e booleanos, e podem ser simples (armazenando um único valor) ou coleções (armazenando múltiplos valores). Na AML, um atributo segue a seguinte estrutura básica:

[modificador] nome: tipo_de_dado [multiplicidade]

Onde o modificador é opcional, utilizando os pictogramas irá definir propriedades especiais do atributo, como chave primária (#), único (-), ou regular (+), caso seja omitido será assumido que é um atributo regular. Nome é o que identifica do atributo dentro da entidade, caso o nome do atributo esteja sublinhado, isso indica que este é um atributo obrigatório. Tipo de dado define a natureza do valor

armazenado, exemplo: *int, str, real, bool*. Multiplicidade é opcional e indica se o atributo pode armazenar um único valor (1) ou múltiplos valores ([*]), caso seja omitido será assumido que o atributo armazena um valor único, a figura 12 ilustra diferentes tipos de atributos.

Figura 12 – Atributos da AML



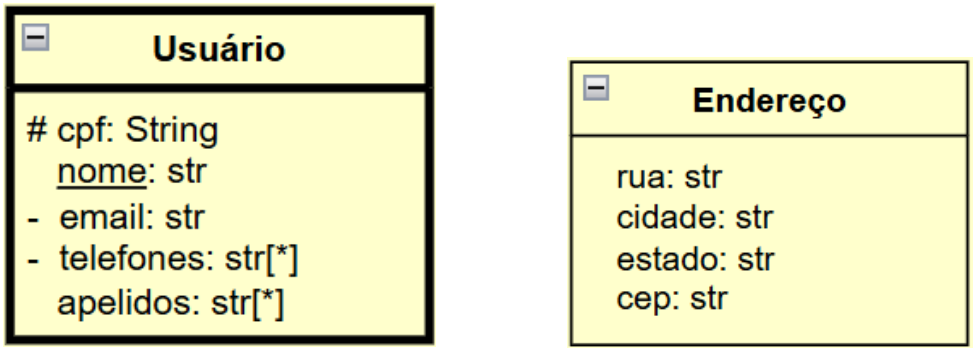
Fonte: Autoria Própria (2025)

Aqui neste exemplo nós lê-se os atributos da seguinte forma:

- **cpf:** Identificador único do usuário, do tipo *string*.
- **nome:** Campo regular obrigatório do tipo *string*.
- **email:** Campo único opcional do tipo *string*.
- **telefones:** Lista de *strings* únicas (*Set*)
- **apelidos:** Lista de *strings* permitindo repetição (Bag)

Seguindo, são apresentados os conceitos de Entidades e Objetos de Valor, ilustrados na figura 13. Conforme foi definido antes, a diferença entre uma entidade e um objeto de valor no modelo de agregados está no fato de que as entidades possuem identidades únicas e os objetos de valor não possuem. Trazendo essa diferença para a representação gráfica da AML, isso significa que os objetos de valor não vão possuir um atributo identificador. Já as entidades, por outro lado, são representadas de forma semelhante às classes ativas da UML, contendo bordas mais grossas ou barras verticais adicionais em suas extremidades. Esse recurso visual tem o objetivo de destacar a entidade no diagrama, diferenciando-a dos objetos de valor, cuja representação é a de uma classe comum com bordas mais finas. A identificação, tanto de entidades quanto de objetos de valor, se dá pelo nome declarado no elemento.

Figura 13 – Entidades e Objetos de Valor da AML

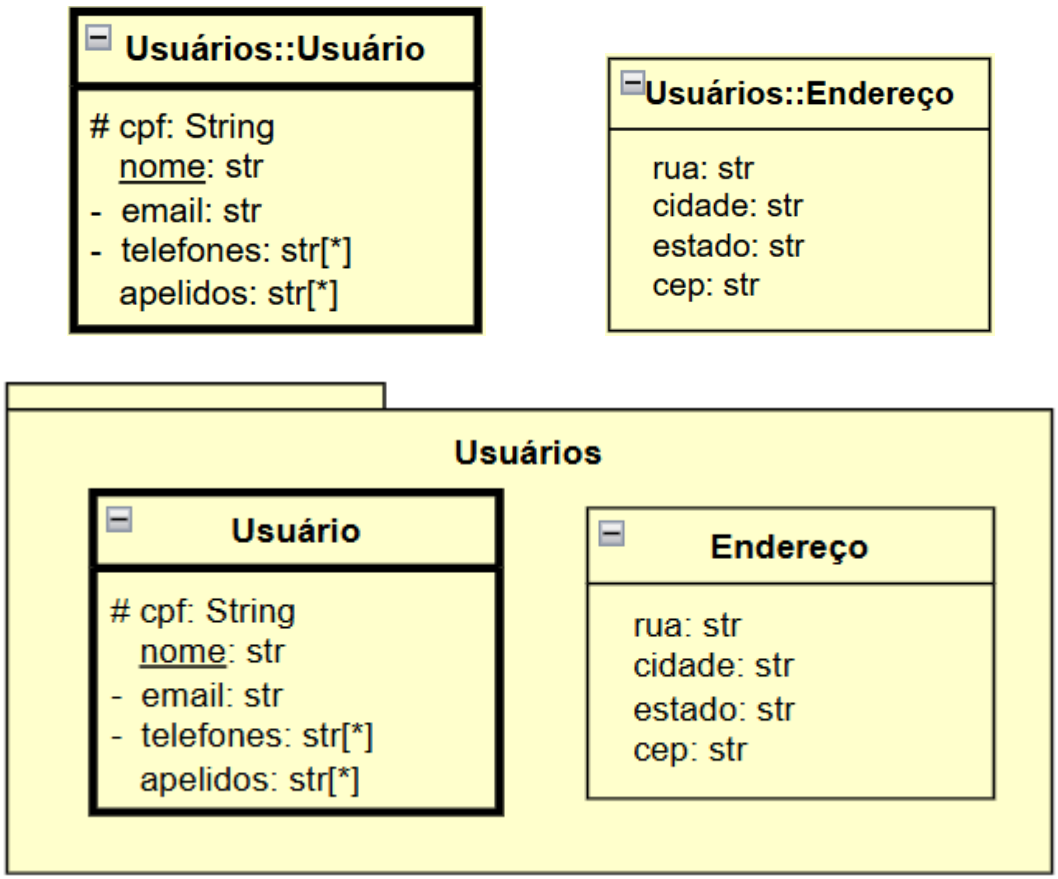


Fonte: Autoria Própria (2025)

Por fim, o conceito mais importante dos nós, o conceito de agregado. Conforme definido anteriormente, um agregado representa um conjunto coeso de entidades e objetos de valor que são tratados como uma unidade lógica dentro do banco de dados, afetando diretamente a eficiência das operações de leitura e escrita. Por isso definir corretamente os limites do agregado é a parte principal da modelagem lógica de um banco de dados de família de colunas. No contexto do HBase, um agregado será um conjunto de informações relacionadas que devem ser armazenadas na mesma linha de uma tabela, agrupando seus atributos dentro de famílias de colunas, tendo como Row Key o atributo identificador da raiz do agregado.

A AML representa os agregados através da representação de pacotes da UML, com isso a representação pode ser feita de duas formas. A primeira delas é adicionar o nome do agregado antes do nome da entidade ou objeto de valor, seguido por " :: " (dois símbolos de dois pontos). Na UML existe também uma segunda forma de representar pacotes que a AML também utiliza para representar seus agregados, que é através do símbolo de pacote. A figura 14 ilustra ambas as notações para representar um agregado Usuários, composto pela entidade Usuário e o Objeto de Valor Endereço.

Figura 14 – Agregados da AML

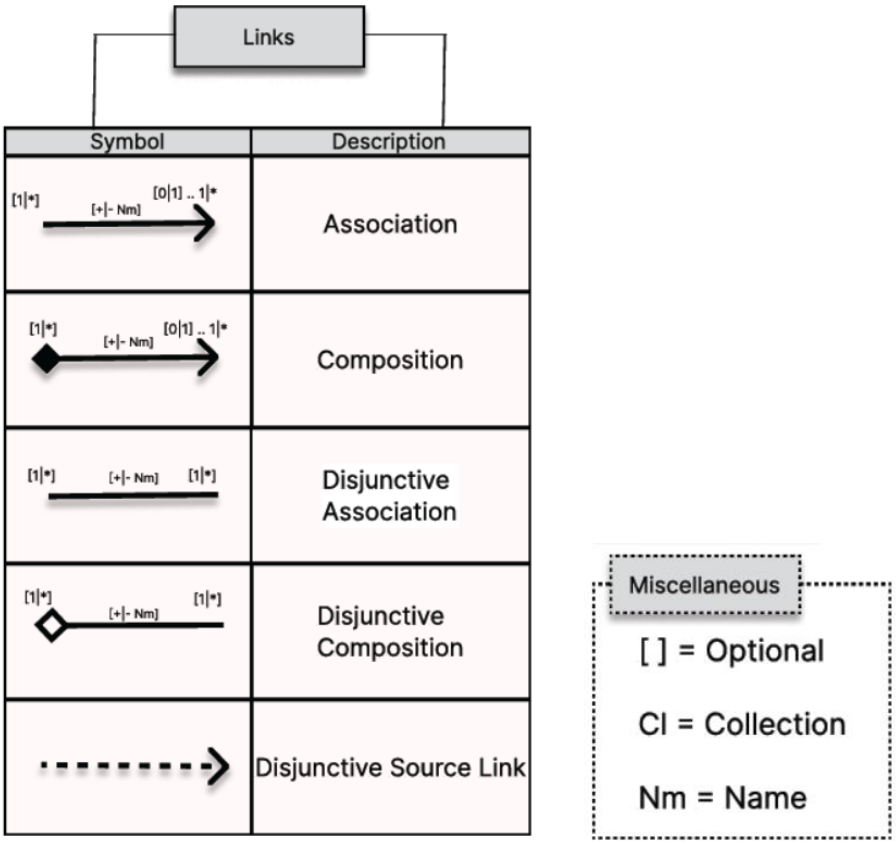


Fonte: Autoria Própria (2025)

3.1.3. Links e Seus Pictogramas

Embora com as informações fornecidas pelos nós já é possível inferir muitas coisas sobre uma determinada modelagem, ainda falta um ponto essencial, que é a forma como os agregados, entidades e objetos de valor se relacionam e como descrever a hierarquia entre eles. Para descrever esses relacionamentos a AML utiliza links, ilustrados na figura 15. A adequada modelagem dos relacionamentos influencia diretamente a organização dos dados e a eficiência das consultas. Os bancos de família de colunas, como o HBase, são otimizados para armazenar dados agrupados dentro de agregados, reduzindo a necessidade de operações custosas de junção.

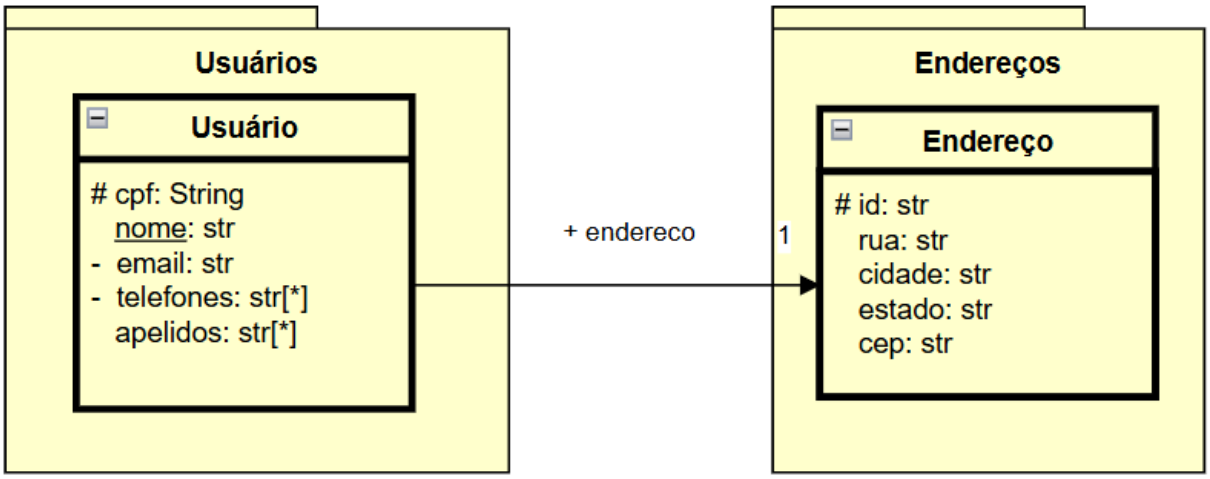
Figura 15 – Links da AML



Fonte: Documentação da AML

Os primeiros dois tipos de links já são bem conhecidos por seguirem os padrões já definidos pela UML, esses são os links de associação e composição. Os links de associação são utilizados quando há uma relação fraca entre as entidades, permitindo que elas existam independentemente. Esse tipo de relacionamento é ideal quando se deseja evitar a redundância de dados, referenciando entidades separadas em colunas específicas. No contexto do HBase, a associação é implementada armazenando a chave da entidade referenciada dentro da linha principal, garantindo que os dados possam ser recuperados de maneira eficiente. Por exemplo, um usuário pode ter um endereço armazenado separadamente e referenciá-lo por meio de um identificador único na coluna *Usuario:endereco*. Essa abordagem permite que múltiplos usuários compartilhem um mesmo endereço, evitando duplicação de informações. Contudo, essa técnica pode exigir múltiplas buscas para recuperar todas as informações relacionadas, o que pode impactar o desempenho dependendo da frequência de acesso (figura 16).

Figura 16 – Link de Associação da AML



Fonte: Autoria Própria (2025)

A figura 16 apresenta um exemplo completo de um caso de modelagem AML. No cenário representado acima o elemento e Endereço deixou de ser um objeto de valor e passou a ser uma entidade, possuindo agora o atributo identificador “id”. Trazendo para o contexto do HBase existirão duas tabelas, cada uma delas com uma família de colunas, ilustradas na figura 17. A tabela que representa o agregado Usuários terá como *Row Id* o CPF dos usuários e a família de colunas será chamada de Usuário. Já a tabela Endereços terá como Row Id o id dos endereços e a família de colunas será chamada de Endereço.

Figura 17 – Exemplo de Tabelas

Tabela Usuários

Row Key (cpf)	Usuario:nome	Usuario:email	Usuario:telefones	Usuario:apelidos	Usuario:endereco
12345678900	João Silva	joao@email.com	["99999", "98888"]	["João", "João"]	E001
98765432100	Maria Souza	maria@email.com	["91111", "97777"]	["Mari", "Souza"]	E002

Tabela Endereços

Row Key (id)	Endereco:rua	Endereco:cidade	Endereco:estado	Endereco:cep
E001	Av. Paulista	São Paulo	SP	01311-000
E002	Rua das Flores	Curitiba	PR	80040-210

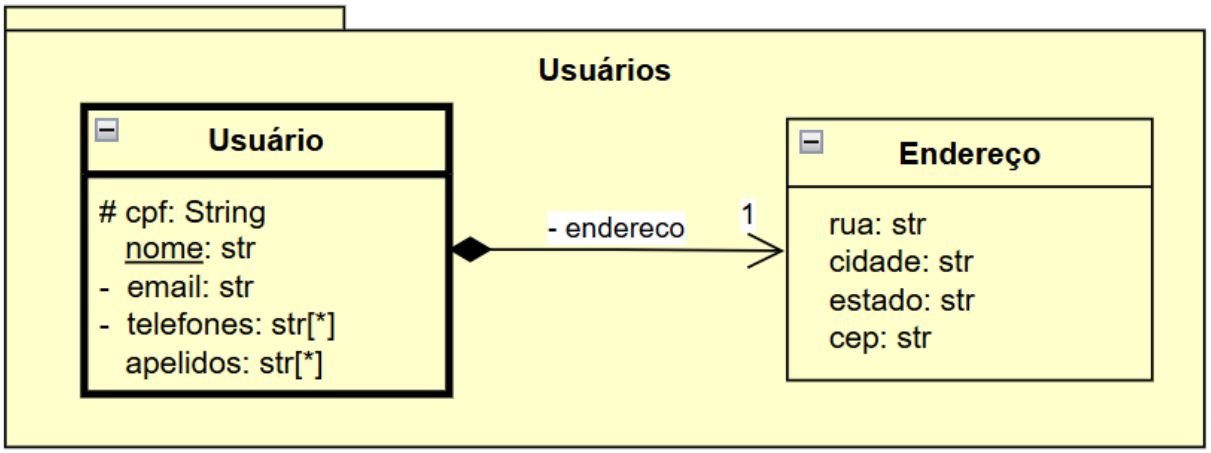
Fonte: Autoria Própria (2025)

É importante salientar que o HBase não fornece mecanismos nativos de integridade referencial, ou seja, não há verificação interna que assegure a existência

ou a atualização desses registros referenciados. Assim, torna-se responsabilidade da aplicação ou de uma camada externa zelar pela consistência dos dados, a fim de evitar inconsistências decorrentes de referências inválidas ou desatualizadas.

Por outro lado, os links de composição representam um relacionamento forte, onde os dados são parte integrante de um único agregado (figura 18). No HBase, isso significa que existirá apenas uma tabela, onde todas as informações de uma entidade e seus componentes são armazenadas na mesma linha, dentro de diferentes famílias de colunas (figura 19). Esse modelo melhora o desempenho de leitura, pois todas as informações podem ser recuperadas em uma única consulta. No caso do usuário e seu endereço, ao invés de armazenar o endereço separadamente, ele pode ser incorporado diretamente dentro do agregado Usuario, garantindo que os dados sejam sempre acessados juntos. Essa abordagem elimina a necessidade de consultas adicionais, mas pode levar à redundância de dados caso muitos usuários compartilhem o mesmo endereço.

Figura 18 – Link de Composição da AML



Fonte: Autoria Própria (2025)

Figura 19 – Exemplo de Tabelas

Tabela Usuários

Row Key (cpf)	Usuario:nome	Usuario:email	Usuario:telefones	Usuario:apelidos	Endereco:rua	Endereco: cidade	Endereco: estado	Endereco:cep
12345678900	João Silva	joao@email.com	["99999", "98888"]	["João", "João"]	Av. Paulista	São Paulo	SP	01311-000
98765432100	Maria Souza	maria@email.com	["91111", "97777"]	["Mari", "Souza"]	Rua das Flores	Curitiba	PR	80040-210

Fonte: Autoria Própria (2025)

Além da associação e composição, a AML também permite a modelagem de relacionamentos usando links de disjunção, que são aplicáveis quando uma

entidade pode estar relacionada, exclusivamente, a diferentes tipos de objetos. Esse conceito é útil quando um mesmo agregado precisa representar diferentes categorias de dados, como no caso de usuários do tipo Pessoa Física e Pessoa Jurídica. Existem três principais tipos de links de disjunção: associação disjuntiva, composição disjuntiva e link de origem disjuntivo. A associação disjuntiva representa um relacionamento onde uma entidade pode se associar a múltiplos tipos sem dependência estrutural, sendo útil quando uma entidade pode referenciar diferentes tipos de objetos sem que sua estrutura seja afetada. Já a composição disjuntiva indica um vínculo, onde diferentes tipos de objetos podem ser agregados dentro da mesma entidade principal, garantindo que todos os dados relevantes sejam armazenados juntos e acessados em uma única leitura. O link de origem disjuntivo é usado para representar referências dinâmicas entre diferentes entidades, funcionando como um direcionamento flexível dentro do modelo. No contexto de bancos de família de colunas, os links de disjunção são valiosos para permitir que uma entidade possa conter atributos variáveis ou associar-se a diferentes categorias, otimizando a modelagem de esquemas flexíveis e evitando estruturas rígidas e pouco escaláveis. Por fim, destacam-se duas observações sobre disjunções. A primeira é que, nos links de Associação Disjuntiva e Composição Disjuntiva, a multiplicidade existe somente no destino e se restringe à multiplicidade máxima. Não se define multiplicidade na fonte do link, pois a disjunção representa uma escolha entre diferentes tipos de entidades e, portanto, vincular multiplicidade mínima ou um range ao lado de origem geraria ambiguidade. Segundo, a multiplicidade mínima não deve ser definida para nenhum Link de Extremidade Disjuntivo, pois uma disjunção representa um relacionamento opcional entre diferentes tipos de elementos, ou seja, a entidade pode estar associada a uma ou mais opções sem exigir uma quantidade mínima fixa.

3.1.4. Conceitos da AML no HBase

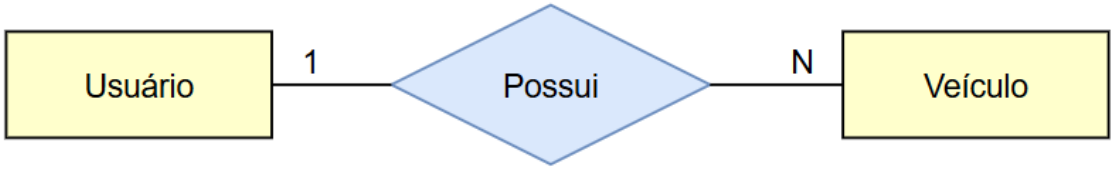
- **Agregado:** No HBase, cada agregado tende a corresponder a uma tabela lógica, pois o agrupamento de dados segue os limites de consistência estabelecidos para aquele conjunto. Cada linha nessa tabela corresponde a uma instância do agregado, usando a *rowkey* para identificar a raiz.

- **Raiz do Agregado:** A *rowkey* da tabela HBase é o atributo identificador da entidade da raiz do agregado. Todas as informações relativas àquele agregado são indexadas por essa chave principal, facilitando leituras e escritas atômicas em nível de linha.
- **Entidade:** Entidades são traduzidas para famílias de colunas no HBase. Caso a entidade seja a raiz do agregado, a entidade estará diretamente associada à *rowkey*.
- **Objeto de Valor:** Mapeado também em famílias de colunas na mesma linha que a entidade à qual pertence
- **Atributo:** Em HBase, cada atributo pode ser uma ou mais colunas dentro de uma família de colunas. Para *arrays*, é comum criar colunas dinâmicas, com colunas adicionadas simulando os elementos do *array* ou usar um campo JSON único.
- **Link de Composição:** Indica que os dados pertencem à mesma linha do HBase, reforçando a ideia de que compõem o mesmo agregado. Assim, os valores são gravados como colunas ou famílias de colunas diretamente atreladas à *rowkey* da entidade raiz.
- **Link de Associação:** Em HBase, essa referência pode ser feita via armazenamento do ID da entidade referenciada em uma coluna. Frequentemente conduz a duas (ou mais) tabelas separadas, cada uma com sua *rowkey*.
- **Aplicação:** Parte dos mecanismos evidenciados na AML, como checagens de unicidade, garantias de obrigatoriedade e integridade referencial, não são fornecidos de forma nativa pelo HBase. Portanto, esses conceitos devem ser implantados na camada de aplicação, seja por meio de verificações antes da inserção ou atualização, seja por meio de estratégias de indexação e validação de dados. Dessa forma, o desenho lógico (com pictogramas de unicidade, obrigatoriedade e integridade de referências) se converte em regras de negócio que a aplicação precisa efetivamente gerenciar para manter a coerência do modelo.

4. APLICAÇÃO DA AML NA MODELAGEM DE BANCOS DE DADOS DE FAMÍLIA DE COLUNAS UTILIZANDO HBASE

Após a apresentação dos conceitos fundamentais da AML nos capítulos anteriores, este capítulo foca na aplicação prática dessas ferramentas. O objetivo é explorar como diferentes estratégias de modelagem podem ser utilizadas para representar um mesmo esquema conceitual, analisando suas diferenças e os impactos que cada abordagem pode ter em um banco de família de colunas como o HBase. Para isso, será tomado como base o cenário ilustrado na figura 20, onde um usuário pode possuir múltiplos veículos, enquanto cada veículo pertence a um único usuário. A partir dessa relação, serão apresentadas diferentes formas de modelá-la. Além de comparar os efeitos práticos de cada abordagem, este capítulo tem como propósito principal demonstrar a completude da AML como linguagem de modelagem lógica para bancos de dados de família de colunas. Ao aplicar seus construtores em cenários distintos, com diferentes formas de embutimento, associação e redundância, busca-se evidenciar que a AML é capaz de representar, com clareza e fidelidade, os principais elementos estruturais do Apache HBase. Cada abordagem será discutida em termos de suas vantagens e desvantagens, considerando fatores como eficiência no armazenamento, facilidade de recuperação dos dados e impacto na escalabilidade. O objetivo deste capítulo é fornecer uma visão comparativa das abordagens possíveis, ajudando a compreender como as escolhas de modelagem podem afetar o desempenho do sistema e quais critérios devem ser levados em conta ao projetar um banco de dados de família de colunas.

Figura 20 – Modelo Conceitual a ser Utilizado



Fonte: Autoria Própria (2025)

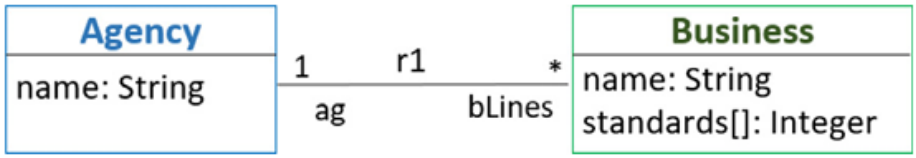
4.1. METODOLOGIA DE AVALIAÇÃO

Para garantir comparabilidade e replicabilidade, cada cenário apresentado seguirá os mesmos critérios de análise, abrangendo descrição, modelagem (diagrama AML), vantagens, desvantagens e impactos nas características CAP. Essa sistemática permite avaliar, de forma coerente, como diferentes escolhas de modelagem afetam o desempenho, a escalabilidade e a coerência dos dados no HBase, além de oferecer uma metodologia que pode ser reproduzida em outros bancos colunares. Dessa maneira, busca-se evidenciar como as fronteiras de agregados, as composições e as associações propostas pela AML podem ser traduzidas em tabelas, famílias de colunas e *row keys* nesse tipo de SGBD distribuído.

4.1.1. Estrutura de Cada Cenário

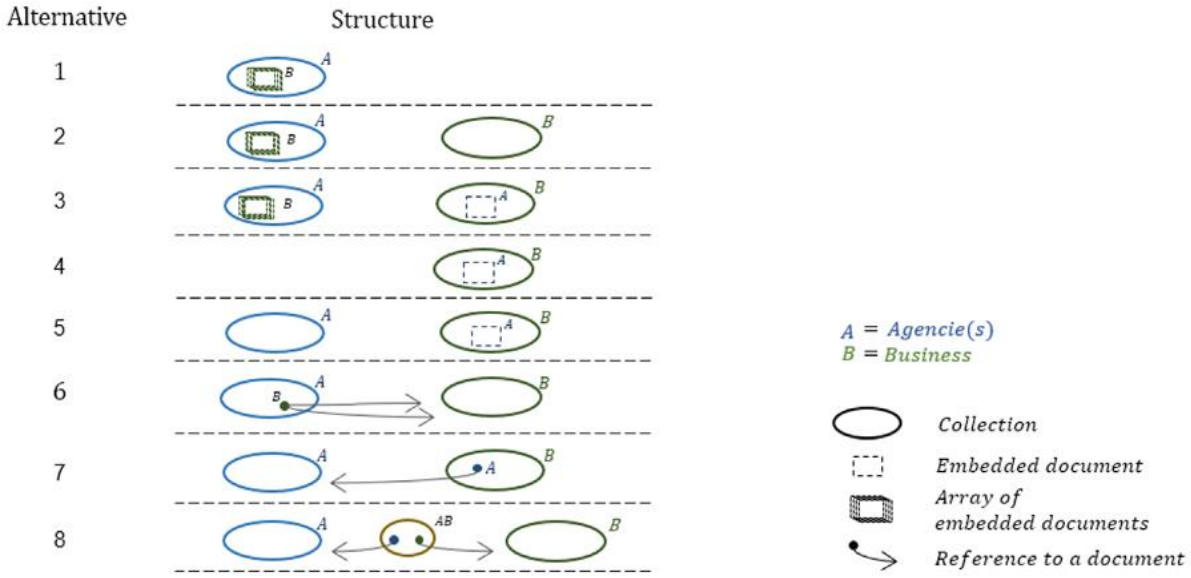
Para facilitar a organização e a análise comparativa das diferentes formas de modelagem, este trabalho adota a mesma estrutura de cenários de modelagem proposta por Gómez et al. (2021). Em tal artigo os autores utilizam a ferramenta *ScorusTool*, a qual recebe como *input* do usuário um diagrama UML do relacionamento a ser modelado (figura 21), e a ferramenta gera automaticamente um conjunto de alternativas de como estruturar esses dados (figura 22).

Figura 21 – Diagrama UML do Relacionamento Agency - Business



Fonte: GÓMEZ (2021)

Figura 22 – Alternativas de Estruturas de Dados



Fonte: GÓMEZ (2021)

O relacionamento *usuário-veículo* analisado no presente estudo apresenta características conceituais e estruturais equivalentes ao relacionamento *agência-negócio* utilizado por Gómez et al. (2021), permitindo que as mesmas variações de composição, associação e redundância sejam aplicadas diretamente. Dessa forma, os oito cenários aqui apresentados seguem a mesma lógica e nomenclatura, contemplando desde estruturas com dados totalmente embutidos até alternativas com múltiplas tabelas e referências cruzadas entre entidades. Na tabela 1, é apresentado uma visão geral dos cenários.

Tabela 1 - Cenários de Modelagem

Cenário	Título	Descrição Geral
1	Uma Tabela com *Array* de Elementos Embutidos	O Usuário (raiz do agregado) contém todos os Veículos dentro de um *array* embutido, armazenado na mesma linha.
2	Uma Tabela com *Array* de Elementos Embutidos e Tabela Redundante	Semelhante ao cenário anterior, porém há redundância controlada: os veículos também são guardados em outra tabela, equilibrando acesso rápido e manutenção adicional.
3	Uma Tabela com *Array* de Elementos Embutidos e Tabela com Apenas um Elemento Embutido	Combinação dos cenários 1 e 2, resultando em redundância total: tanto os dados do usuário quanto dos veículos são duplicados em tabelas distintas, otimizando leituras, mas exigindo alto cuidado de sincronização.
4	Uma Tabela com Apenas um Elemento Embutido	Inversão do cenário 1: agora o Veículo é a raiz e embute as informações do Usuário, útil quando o acesso frequente se concentra nos detalhes de cada veículo.
5	Uma Tabela com Apenas um Elemento Embutido e Tabela Redundante	Variante do cenário 4 com redundância parcial: os dados do usuário ficam no próprio veículo, enquanto outra parte (ou tudo) é replicada em uma tabela dedicada a usuários.
6	Duas Tabelas com *Array* de Elementos Referenciados	O Usuário mantém um *array* de referências (IDs) para seus Veículos, que são armazenados numa tabela separada sem embutir dados redundantes.
7	Duas Tabelas com Apenas um Elemento Referenciado	Similar ao anterior, porém cada Veículo possui apenas o ID do Usuário; não há duplicação de dados, mas a leitura conjunta de usuário e veículo exige várias consultas.
8	Três Tabelas Sendo uma Intermediária de Referências	Abordagem que separa as entidades (Usuários e Veículos) e adiciona uma tabela intermediária para mapear o relacionamento, eliminando redundância ao custo de mais operações.

Fonte: Autoria Própria (2025)

Para cada um desses cenários, foi adotada a seguinte estrutura de análise, a fim de manter a comparabilidade e a replicabilidade do estudo:

- **Descrição do Cenário**

Apresenta-se uma visão geral do relacionamento (usuários e veículos) e como os dados se interligam.

- **Modelagem AML**

Mostra-se, por meio da *Aggregate Modeling Languag*e, como as entidades são estruturadas (composição vs. associação), incluindo atributos relevantes e definição dos agregados.

- **Pontos Positivos**

Enfatizam-se os benefícios em termos de redução de consultas, atomicidade de atualizações, facilidade de leitura, escalabilidade ou qualquer outro ganho proporcionado pela abordagem.

- **Pontos Negativos**

Discutem-se os efeitos colaterais, como redundância de dados, maior complexidade de sincronização ou necessidade de múltiplas leituras, entre outros desafios de manutenção.

- **Impactos nas Características CAP**

Avaliam-se as implicações de cada forma de modelagem no HBase, considerando a consistência em nível de linha, a disponibilidade em cenários de falha e as estratégias de particionamento.

- **Estrutura de Tabelas e Exemplos de Consultas no HBase**

Quando pertinente, serão apresentados comandos de *Get* ou *Scan* para ilustrar a complexidade de recuperar informações em cada cenário.

4.2. CENÁRIOS DE MODELAGEM

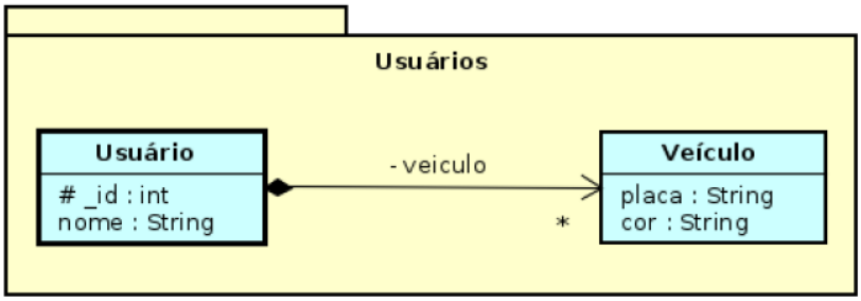
4.2.1. Uma Tabela com *Array* de Elementos Embutidos

1- Descrição do Cenário Utilizando AML

Neste primeiro cenário tem-se um único agregado no qual o Usuário é a raiz do agregado, e os Veículos estão contidos dentro dele como colunas dinâmicas. Isso significa que os dados de um usuário e de seus veículos são armazenados juntos na mesma linha dentro da tabela Usuários no HBase. Essa estrutura garante que, ao recuperar os dados de um usuário, todos os veículos associados a ele serão

acessados na mesma leitura, eliminando a necessidade de múltiplas consultas. Essa estratégia melhora a eficiência das leituras, pois todos os dados do agregado são armazenados fisicamente próximos dentro dos mesmos blocos de disco. Conforme ilustrado na figura 23, neste cenário com apenas um agregado, um veículo só pode ser consultado dentro do contexto do usuário ao qual ele pertence. Isso restringe o acesso direto aos veículos, o que pode ser uma vantagem quando se deseja garantir maior consistência no modelo de dados e evitar alterações isoladas que possam comprometer a integridade das informações.

Figura 23 – Cenário de Usuário Contendo Vários Veículos



Fonte: MONTEIRO (2023)

Na figura 23 apresenta-se o nó Usuários identificado pela raiz do agregado, a entidade Usuário sendo identificada pelo campo _id e possuindo o atributo nome do tipo *String,* e o Objeto de Valor Veículo com os atributos placa e cor. Para finalizar a modelagem tem-se o link de composição, indicando que um usuário pode ter vários veículos embutidos, porém sem repetições (figura 23). Nessa estrutura, os veículos não existem de forma independente; eles integram o agregado do usuário, tornando a instância de Usuario responsável por criar, manter e remover os registros de Veiculo. Com isso, as operações transacionais ocorrem sobre o agregado como um todo, garantindo atomicidade nas modificações.

2- Pontos Positivos

- **Leituras Unificadas:** Todas as informações do usuário e de seus veículos são obtidas em um único acesso ao HBase, reduzindo a latência das consultas.

- **Consistência de Dados:** Eventuais alterações são aplicadas em nível de linha, o que evita problemas de concorrência ao atualizar tanto o usuário quanto os veículos.
- **Escalabilidade em Acesso Baseado no Usuário:** Sempre que a aplicação necessita de dados de um usuário e de todos os seus veículos, o cenário é altamente otimizado, melhorando o *throughput* de leitura.
- **Facilidade de Manutenção no Contexto do Agregado:** Como há apenas uma tabela, a lógica de persistência se torna menos complexa para operações que envolvem o objeto usuário e sua lista de veículos.

3- Pontos Negativos

- **Acesso Independente aos Veículos**: Não há como consultar diretamente um veículo sem antes recuperar o usuário ao qual ele pertence. Em cenários que exigem buscar veículos isolados, podem ser necessárias múltiplas leituras ou outra estratégia adicional.
- **Possível Crescimento Exagerado da Linha**: Se um usuário possuir muitos veículos, o volume de dados embutidos em uma única linha pode afetar negativamente o desempenho, sobretudo em cenários de escrita ou atualização frequente.
- **Flexibilidade Reduzida**: Se a aplicação evoluir para cenários em que os veículos precisem ser consultados e atualizados de forma independente, esse modelo perde eficiência.

4- Impactos nas Características CAP do HBase

- **Consistência:** O HBase aplica consistência em nível de linha, garantindo que as alterações em um usuário e seus veículos sejam atômicas. Desse modo, a integridade interna do agregado é facilmente assegurada.
- **Disponibilidade**: Em casos de falha de região *(region server)*, tanto o usuário quanto seus veículos podem ficar temporariamente indisponíveis até que a região seja realocada. Entretanto, essa é uma característica inerente ao HBase, não sendo exclusiva desse modelo.

- **Particionamento**: O particionamento se dá pela chave da linha (_id). Isso significa que se houver concentração de muitas operações sobre determinados usuários, pode ter a sobrecarga de poucas regiões. Ainda assim, o HBase oferece mecanismos para distribuir chaves e amenizar pontos únicos de gargalo.

5- Exemplos de Tabelas e Consultas no HBase

No HBase, não é possível ter um *array* de famílias de colunas embutidas, pois a estrutura do banco não suporta famílias de colunas dinâmicas. As famílias de colunas devem ser definidas estaticamente no momento da criação da tabela e não podem ser criadas dinamicamente como colunas individuais. Entretanto, dentro de uma única família de colunas, é possível ter múltiplas colunas dinâmicas, o que permite simular a estrutura de um *array* embutido. Com isso existirá uma tabela com as famílias de colunas Usuário e Veículos, cada entrada do *array* de veículos é armazenada como uma nova coluna separada identificada pelo índice do *array*, conforme ilustrado na figura 24. Também é possível armazenar o *array* em formato JSON dentro de uma única célula, este cenário se aplica mais quando os dados não precisam ser acessados individualmente.

Figura 24 – Exemplo de Tabelas e Consultas do Cenário 1

Tabela Usuários

Row Key (id)	Usuario:nome	Veiculos:0:placa	Veiculos:0:cor	Veiculos:1:placa	Veiculos:1:cor
1	João Silva	ABC1234	Azul	DEF5678	Preto
2	Maria Souza	GHI9012	Vermelho	JKL3456	Branco

```
get 'Usuarios', '1'

COLUMN                          CELL
Usuario:nome                    timestamp=..., value=Joao Silva
Veiculos:1:placa                timestamp=..., value=ABC1234
Veiculos:1:cor                  timestamp=..., value=Azul
Veiculos:2:placa                timestamp=..., value=DEF5678
Veiculos:2:cor                  timestamp=..., value=Preto
```

```
get 'Usuarios', '1', 'Veiculos:2:cor'

COLUMN                            CELL
Veiculos:2:cor                    timestamp=..., value=Preto
```

Fonte: Autoria Própria (2025)

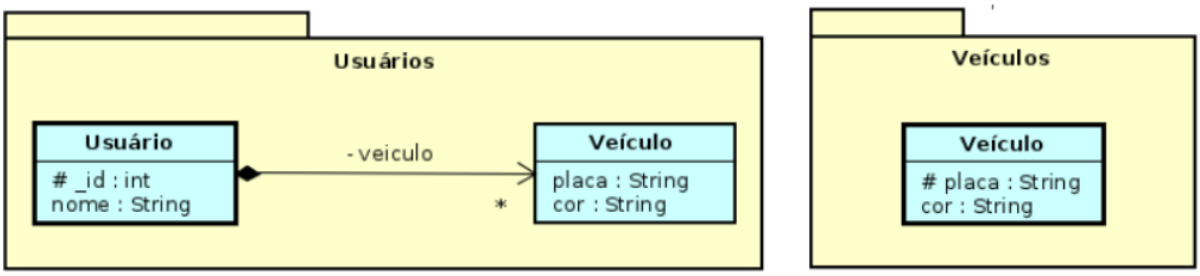
4.2.2. Uma Tabela com *Array* de Elementos Embutidos e Tabela Redundante

1- Descrição do Cenário Utilizando AML

Neste segundo cenário, adota-se uma abordagem híbrida para armazenar os dados de Usuários e Veículos. Por um lado, cada usuário mantém internamente um *array* de veículos, preservando o benefício de leituras eficientes quando se deseja recuperar todos os veículos daquele usuário em uma única operação. Ao mesmo tempo, que se introduz uma redundância controlada ao criar-se uma tabela separada para os veículos, viabilizando o acesso independente a cada registro de veículo sem a necessidade de percorrer toda a tabela de usuários (figura 25). Essa estratégia tem como principal motivação conciliar a eficiência de leitura, típica de modelos com dados embutidos, com a flexibilidade de acesso que surge ao dissociar a entidade veículo do contexto estritamente vinculado ao usuário. No entanto, esse ganho de acessibilidade traz consigo um aumento de redundância: o mesmo veículo fica armazenado duas vezes, tanto dentro do agregado do usuário

quanto na tabela específica de veículos. Essa duplicação pode impactar o consumo de armazenamento e ampliar a complexidade de sincronização, pois cada mudança (adição, remoção ou atualização de um veículo) exige operações de escrita em ambas as estruturas.

Figura 25 – Cenário de Usuário Contendo Vários Veículos e Redundância de Veículos



Fonte: MONTEIRO (2023)

Conforme ilustrado na figura 25, neste caso existem dois agregados, um sendo o nó Usuários idêntico ao cenário anterior e um novo agregado representado pelo nó Veículos (figura 25). Este novo agregado será composto apenas da entidade Veículo, tendo o atributo placa do tipo *string* como identificador e o atributo regular cor. Este cenário não possui links de relacionamento entre os agregados, pois Veículos é introduzido como redundância das informações embutidas em Usuários. A duplicação de dados decorre do fato de que o mesmo veículo aparece em dois lugares: no *array* embutido dentro do usuário e, simultaneamente, em sua tabela própria.

2- Pontos Positivos

- **Leituras Rápidas no Contexto do Usuário:** Em consultas que demandam todos os veículos de um usuário, o *array* embutido elimina a necessidade de múltiplas operações de leitura.
- **Acesso Independente ao Veículo:** A existência de uma tabela exclusiva para veículos possibilita buscas diretas pela placa do veículo ou por atributos específicos de forma mais eficiente, sem ler toda a linha de usuário.
- **Balanceamento de Carga:** Podem-se realizar leituras frequentes dos detalhes do veículo na tabela dedicada, reduzindo o stress sobre a tabela de

usuários, que continua otimizada para operações que se baseiam no _id do usuário.

3- Pontos Negativos

- **Redundância de Dados:** O armazenamento duplicado dos veículos aumenta o consumo de espaço. Em cenários de alta cardinalidade, essa redundância se torna ainda mais significativa.
- **Sincronização Complexa:** Sempre que um veículo é adicionado, removido ou alterado, duas operações de escrita precisam ser executadas — uma na tabela de usuários e outra na tabela de veículos. Esse requisito dobra a carga de escrita no HBase e abre margem para inconsistências caso haja falha parcial na gravação.
- **Maior Complexidade de Desenvolvimento:** A aplicação precisa gerenciar a lógica de atualização em duas estruturas, garantindo que os dados embutidos no usuário sejam coerentes com aqueles armazenados na tabela de veículos.

4- Impactos nas Características CAP do HBase

- **Consistência:** A consistência interna de cada linha permanece assegurada no HBase, mas a duplicidade exige práticas de sincronização para garantir que as alterações em um veículo reflitam corretamente em ambos os locais.
- **Disponibilidade:** Em geral, mantém-se a disponibilidade de leitura tanto no agregado de usuário quanto na tabela de veículos. Contudo, se a região que hospeda a tabela de usuários estiver indisponível, só será possível consultar informações do veículo na tabela separada, e vice-versa.
- **Particionamento:** Cada agregado permanece particionado de acordo com a chave primária de sua respectiva tabela, _id para a tabela de usuários e placa para a tabela de veículos. Em situações de partição de rede, uma parte do cluster pode atualizar usuários na tabela independente enquanto outra atualiza os veículos, gerando discrepâncias temporárias.

5- Exemplos de Tabelas e Consultas no HBase

Para este cenário uma possibilidade de implementação é ter duas tabelas no HBase, uma tabela Usuários igual ao cenário anterior, contendo duas famílias de colunas. A primeira família sendo a família Usuário, com a informação do nome, e a segunda sendo a família Veículos, com um *array* embutido, onde cada posição possui informações de placa e cor. E uma segunda tabela Veículos, que seria a tabela redundante, a qual teria a placa como *Row Key* e a família de colunas Veículo com as informações de cor (figura 26).

Figura 26 – Exemplo de Tabelas e Consultas do Cenário 2

Tabela Usuários

Row Key (id)	Usuario:nome	Veiculos:0:placa	Veiculos:0:cor	Veiculos:1:placa	Veiculos:1:cor
1	João Silva	ABC1234	Azul	DEF5678	Preto
2	Maria Souza	GHI9012	Vermelho	JKL3456	Branco

Tabela Veículos

Row Key (placa)	Veiculo:cor
ABC1234	Azul
DEF5678	Preto
GHI9012	Vermelho
JKL3456	Branco

```
get 'Usuarios', '1'

COLUMN                    CELL
Usuario:nome              timestamp=..., value=Joao Silva
Veiculos:1:placa          timestamp=..., value=ABC1234
Veiculos:1:cor            timestamp=..., value=Azul
Veiculos:2:placa          timestamp=..., value=DEF5678
Veiculos:2:cor            timestamp=..., value=Preto
```

```
get 'Veiculos', 'ABC1234'

COLUMN                       CELL
Veiculo:cor                timestamp=..., value=Azul
```

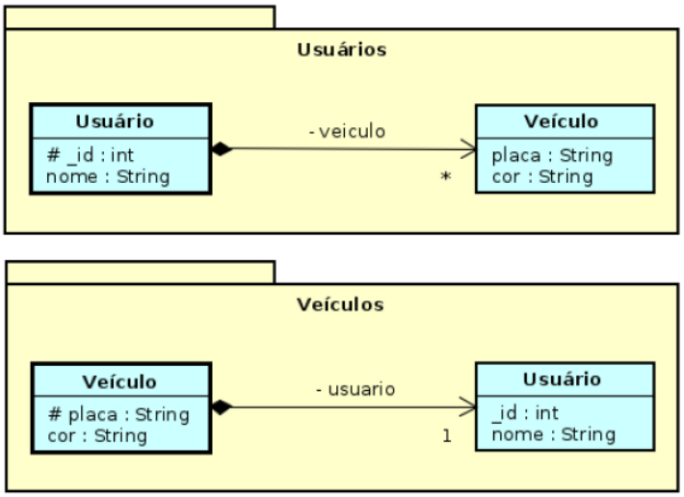
Fonte: Autoria Própria (2025)

4.2.3. Uma Tabela com *Array* de Elementos Embutidos e Tabela com Apenas um Elemento Embutido

1- Descrição do Cenário Utilizando AML

Neste cenário tem-se uma combinação dos dois primeiros, aqui o modelo introduz redundância total, pois tanto os usuários quanto os veículos são armazenados duas vezes de maneira completa, tanto dentro da linha do usuário (como um *array* de veículos embutidos na tabela Usuarios) quanto em uma tabela separada Veiculos, onde cada veículo armazena a relação com um único usuário, como ilustrado na figura 27. Essa abordagem mantém os benefícios de recuperação rápida de veículos por usuário que visto nos cenários anteriores, ao mesmo tempo em que permite a recuperação independente de veículos, mas com o custo adicional de duplicação total dos dados. A principal motivação dessa modelagem é garantir que todas as consultas tragam informações completas sem a necessidade de múltiplas buscas, mas isso pode impactar armazenamento, consistência e complexidade de manutenção. Esse modelo é extremamente eficiente para consultas, pois permite recuperar dados completos sem necessidade de buscas adicionais. No entanto, a duplicação total das informações exige cuidados extras na sincronização entre as tabelas. Essa abordagem é recomendada para sistemas que priorizam leituras rápidas e acessos diretos, mas pode se tornar um problema em cenários de alta escalabilidade, onde a atualização frequente de dados pode sobrecarregar o sistema.

Figura 27 – Cenário de Redundância Total com Usuário Contendo Vários Veículos e Veículo Contendo Um Usuário



Fonte: MONTEIRO (2023)

Na figura 27 é possível identificar dois agregados novamente, um sendo o nó Usuários idêntico ao primeiro cenário e um segundo agregado representado pelo nó Veículos. Este segundo agregado terá com raiz a entidade Veículo, tendo o atributo placa do tipo *string* como identificador e o atributo regular cor, só que diferente do cenário anterior o agregado Veículos possui também o *Value Object* Usuário embutido, composto pelo atributo de _id e nome. E por fim tem-se o link de composição entre os dois, identificando que cada Veículo só pertence a um Usuário.

2- Pontos Positivos

- **Leituras Otimizadas**: Caso a consulta seja orientada ao usuário, a tabela de usuários já possui todos os veículos embutidos. Se a consulta for pautada no veículo, a tabela de veículos armazena as informações do usuário na mesma linha.
- **Foco Contextual Direto**: Não importa se a aplicação está lidando com um ou vários veículos, ou se precisa de detalhes completos do usuário, qualquer cenário dispensa leituras adicionais, pois os dados são recuperados em uma única operação.
- **Redução de Latência**: Com a informação armazenada em duplicidade, evita-se múltiplas leituras no HBase, promovendo um tempo de resposta menor para consultas completas.

3- Pontos Negativos

- **Duplicação Total dos Dados:** Tanto os usuários quanto os veículos acabam sendo armazenados em duas tabelas, com dados repetidos em dois sentidos. Esse fator pode levar a um consumo de armazenamento significativamente maior.
- **Sincronização Complexa:** O processo de escrita se torna mais intrincado. Uma atualização em qualquer atributo de usuário ou de veículo obriga a aplicação a manter esses dados sincronizados em múltiplos locais. Falhas de rede ou sistema durante a escrita podem resultar em inconsistências.

- **Sobrecarga em Grandes Escalas:** Ambientes de grande volume de dados podem ser onerados pelo custo de manter diversas cópias sincronizadas, principalmente se existirem muitos relacionamentos ou alta frequência de atualizações.

4- Impactos nas Características CAP do HBase

- **Consistência:** A consistência em nível de linha permanece garantida dentro de cada tabela do HBase. Todavia, a consistência entre as tabelas é responsabilidade da camada de aplicação, já que a duplicação total exige múltiplas operações de escrita.
- **Disponibilidade:** Cada agregado (usuarios, veiculos) tem sua própria disponibilidade, pois estão em tabelas separadas. Uma falha na região que contém a tabela de usuários não impede a consulta aos veículos na outra tabela, e vice-versa.
- **Particionamento:** As chaves de linha determinam como cada tabela é particionada. A duplicação de dados distribui a carga de forma relativa às consultas, mas exige que se gerencie cuidadosamente a escalabilidade para não concentrar volume excessivo de informações em algumas regiões.

5- Exemplos de Tabelas e Consultas no HBase

Para este cenário uma possibilidade de implementação é ter duas tabelas no HBase, uma tabela Usuários igual ao primeiro cenário, contendo duas famílias de colunas. A primeira família sendo a família Usuário, com a informação do nome, e a segunda sendo a família Veículos, com um *array* embutido, onde cada posição possui informações de placa e cor. E uma segunda tabela Veículos, que seria a tabela redundante, a qual teria a placa como *Row Key* e a família de colunas Veículo com as informações de cor, e diferente do cenário anterior, esta segunda tabela teria uma segunda família de colunas, chamada de Usuário, contendo as colunas de id e nome (figura 28).

Figura 28 – Exemplo de Tabelas e Consultas do Cenário 3

Tabela Usuários

Row Key (id)	Usuario:nome	Veiculos:0:placa	Veiculos:0:cor	Veiculos:1:placa	Veiculos:1:cor
1	João Silva	ABC1234	Azul	DEF5678	Preto
2	Maria Souza	GHI9012	Vermelho	JKL3456	Branco

Tabela Veículos

Row Key (placa)	Veiculo:cor	Usuario:id	Usuario:nome
ABC1234	Azul	1	João Silva
DEF5678	Preto	1	João Silva
GHI9012	Vermelho	2	Maria Souza
JKL3456	Branco	2	Maria Souza

```
get 'Usuarios', '1'

COLUMN                          CELL
Usuario:nome                    timestamp=..., value=Joao Silva
Veiculos:1:placa                timestamp=..., value=ABC1234
Veiculos:1:cor                  timestamp=..., value=Azul
Veiculos:2:placa                timestamp=..., value=DEF5678
Veiculos:2:cor                  timestamp=..., value=Preto
```

```
get 'Veiculos', 'ABC1234'

COLUMN                            CELL
Veiculo:cor                     timestamp=..., value=Azul
Usuario:id                      timestamp=..., value=1
Usuario:nome                    timestamp=..., value=Joao Silva
```

Fonte: Autoria Própria (2025)

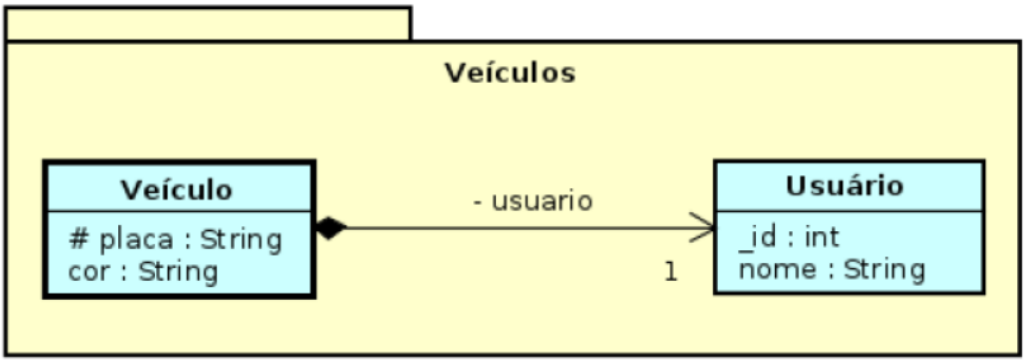
4.2.4. Uma Tabela com Apenas um Elemento Embutido

1- Descrição do Cenário Utilizando AML

Aqui é observado um cenário similar a estrutura do primeiro cenário de modelagem, caso em que não tem redundância de informações e tudo está contido em um único agregado. Porém a orientação dos dados é inversa, aqui o veículo que

possui a raiz do agregado (figura 29). Este cenário é indicado para cenários em que a regra de negócio irá executar leituras focadas nos veículos e não nos usuários, visto que para recuperar todos os veículos de um usuário mais leituras precisarão ser feitas do que no primeiro exemplo.

Figura 29 – Cenário de Veículo Contendo Um Usuário



Fonte: MONTEIRO (2023)

A figura 29 ilustra o inverso do primeiro cenário, o nó Veículos representando o agregado, a entidade Veículo sendo identificada pelo campo placa e possuindo o atributo cor do tipo *String,* e o Objeto de Valor Usuário com os atributos placa e cor. Para finalizar a modelagem tempos o link de composição, indicando que um veículo pode pertencer a apenas um usuário. Nessa estrutura, o usuário é armazenado diretamente dentro do agregado Veículo, não existe mais uma tabela de usuários principal. Cada veículo carrega em seu registro os dados fundamentais do usuário que o utiliza.

2- Pontos Positivos

- **Acesso Otimizado ao Veículo:** As leituras são extremamente rápidas quando a aplicação deseja exibir ou manipular informações associadas a um veículo específico, pois todos os detalhes do usuário necessário para aquela consulta encontram-se no mesmo registro.
- **Consistência Local:** O HBase oferece consistência em nível de linha, de modo que uma atualização no veículo e nos dados do usuário embutido ocorre de forma atômica.

- **Simplicidade de Estrutura:** Apenas uma tabela é necessária para manter os dados primários do negócio (veículos e seus respectivos usuários), reduzindo a complexidade de administração no banco.

3- Pontos Negativos

- **Dificuldade de Recuperar Todos os Veículos de um Usuário:** Para montar a “lista de veículos” de um usuário, é preciso vasculhar toda a tabela de veículos, efetuando leituras potencialmente custosas.
- **Complexidade de Atualização do Usuário:** Caso seja necessário alterar algum atributo do usuário (por exemplo, o nome), é preciso atualizar cada veículo relacionado a ele, o que pode resultar em maior carga de escrita se um usuário possuir muitos veículos.
- **Possibilidade de Crescimento de Linhas:** Assim como no cenário anterior, não há redundância de dados em termos de várias tabelas. No entanto, se um único usuário tiver muitos veículos, a busca agregada pelos dados desse usuário em todos os veículos se torna ineficiente.

4- Impactos nas Características CAP do HBase

- **Consistência:** O HBase assegura que qualquer alteração no veículo e no usuário embutido seja atomicamente aplicada na mesma linha. Porém, se a aplicação manipular muitos veículos associados a um mesmo usuário, as atualizações precisam ocorrer em várias linhas, criando pontos potenciais de inconsistência temporária.
- **Disponibilidade:** As consultas focadas no veículo permanecem altamente disponíveis, pois cada registro é independente. Em caso de falha de região, apenas os veículos contidos nessa região ficam indisponíveis, enquanto o restante do sistema continua operacional.
- **Particionamento:** O particionamento ocorre pela placa do veículo. Esse modelo pode facilitar o balanceamento de carga quando a maioria das

consultas se concentra na individualidade de cada veículo, mas dificulta agrupar dados por usuário.

5- Exemplos de Tabelas e Consultas no HBase

Uma possível implementação no HBase é implementar-se apenas uma tabela, onde a chave de cada linha é a placa do veículo, esta tabela possui duas famílias de colunas. A primeira é Veículo e possui as informações de cor, a segunda é a família de colunas Usuário, com as colunas de id e nome (figura 30). Nesse contexto, a tabela de Usuários deixa de existir como entidade independente, e cada veículo carrega os dados do usuário necessário para suas operações. Essa modelagem favorece sistemas que priorizam leituras pelo veículo, mas acrescenta overhead quando é preciso relacionar diversos veículos ao mesmo usuário.

Figura 30 – Exemplo de Tabelas e Consultas do Cenário 4

Tabela Veículos

Row Key (placa)	Veiculo:cor	Usuario:id	Usuario:nome
ABC1234	Azul	1	João Silva
DEF5678	Preto	1	João Silva
GHI9012	Vermelho	2	Maria Souza
JKL3456	Branco	2	Maria Souza

```
get 'Veiculos', 'ABC1234'

COLUMN                          CELL
Veiculo:cor                     timestamp=..., value=Azul
Usuario:id                      timestamp=..., value=1
Usuario:nome                    timestamp=..., value=Joao Silva
```

```
scan 'Veiculos', {FILTER => "SingleColumnValueFilter('Usuario', 'id', =, 'binary:1')"}


ROW                             COLUMN                          CELL
ABC1234                         Veiculo:cor                     value=Azul
.                               Usuario:id                      value=1
.                               Usuario:nome                    value=Joao Silva
DEF5678                         Veiculo:cor                     value=Preto
.                               Usuario:id                      value=1
.                               Usuario:nome                    value=Joao Silva
```

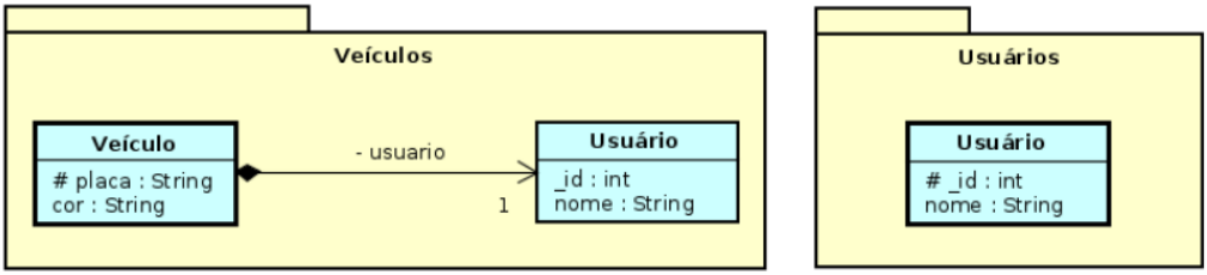
Fonte: Autoria Própria (2025)

4.2.5. Uma Tabela com Apenas um Elemento Embutido e Tabela Redundante

1- Descrição do Cenário Utilizando AML

Este cenário segue a mesma lógica do anterior, em que cada Veículo possui embutidos os dados de Usuário, mas acrescenta uma tabela redundante para permitir acesso independente a cada usuário. Dessa forma, toda a informação do usuário é replicada em dois lugares: Na linha do veículo, que contém o usuário embutido e na tabela exclusiva de usuários, onde cada usuário tem seu próprio registro, conforme ilustra a figura 31. A motivação para esse modelo é fornecer consultas rápidas tanto pelo veículo quanto pelo usuário. Em contrapartida, essa redundância gera custos extras na escrita e na sincronização, pois, sempre que um usuário for alterado, será necessário atualizar tanto a linha correspondente na tabela de usuários quanto os veículos associados.

Figura 31 – Cenário de Veículo Contendo Um Usuário



Fonte: MONTEIRO (2023)

Neste caso existirão dois agregados, um sendo o nó Veículos idêntico ao cenário anterior e um novo agregado representado pelo nó Usuários (figura 31). Este novo agregado será composto apenas da entidade Usuário, tendo o atributo _id do tipo *int* como identificador e o atributo regular nome. Este cenário não possui links de relacionamento entre os agregados, pois Usuários é introduzido como redundância das informações embutidas em Veículos. A duplicação de dados decorre do fato de que o mesmo usuário aparece em dois lugares: embutido dentro do veículo e, simultaneamente, em sua tabela própria.

2- Pontos Positivos

- **Leituras Rápidas pelo Veículo:** Com o usuário embutido na linha do veículo, as consultas focadas na placa ou no modelo retornam também os dados do usuário em uma única operação.
- **Acesso Independente ao Usuário:** A tabela exclusiva de usuários permite buscas por atributos de usuário sem precisar navegar pelas linhas de veículos.
- **Balanceamento de Carga:** Podem-se realizar leituras frequentes dos detalhes do usuário na tabela dedicada, reduzindo o stress sobre a tabela de veículos, que continua otimizada para operações que se baseiam na placa do veículo.

3- Pontos Negativos

- **Redundância de Dados:** Assim como em cenários híbridos anteriores, manter os dados do usuário duplicados consome mais espaço e impõe maior complexidade de escrita.
- **Sincronização Complexa:** Qualquer alteração no usuário requer duas (ou mais) operações de escrita: uma na tabela de usuários e outra em todos os veículos associados ao usuário, sob risco de inconsistências.
- **Maior Complexidade de Desenvolvimento:** A aplicação precisa gerenciar a lógica de atualização em duas estruturas, garantindo que os dados embutidos no veículo sejam coerentes com aqueles armazenados na tabela de usuários.

4- Impactos nas Características CAP do HBase

- **Consistência:** A consistência interna de cada linha permanece assegurada no HBase, mas a duplicidade exige práticas de sincronização para garantir que as alterações em um usuário reflitam corretamente em ambos os locais.
- **Disponibilidade:** Em geral, mantém-se a disponibilidade de leitura tanto no agregado de veículos quanto na tabela de usuários. Contudo, se a região que

hospeda a tabela de veículos estiver indisponível, só será possível consultar informações do usuário na tabela separada, e vice-versa.

- **Particionamento:** Cada agregado permanece particionado de acordo com a chave primária de sua respectiva tabela, _id para a tabela de usuários e placa para a tabela de veículos. Em situações de partição de rede, uma parte do cluster pode atualizar usuários na tabela independente enquanto outra atualiza os veículos, gerando discrepâncias temporárias.

5- Exemplos de Tabelas e Consultas no HBase

Para este cenário uma possibilidade de implementação é ter duas tabelas no HBase, uma tabela Veículos igual ao cenário anterior, com a placa como *Row Key*, contendo duas famílias de colunas. A primeira família sendo a família Veículo, com a informação da cor, e a segunda sendo a família Usuário, onde cada coluna possui informações de id e nome. E uma segunda tabela Usuários, que seria a tabela redundante, a qual teria o id como *Row Key* e a família de colunas Usuário com as informações de nome (figura 32).

Figura 32 – Exemplo de Tabelas e Consultas do Cenário 5

Tabela Veículos

Row Key (placa)	Veiculo:cor	Usuario:id	Usuario:nome
ABC1234	Azul	1	João Silva
DEF5678	Preto	1	João Silva
GHI9012	Vermelho	2	Maria Souza
JKL3456	Branco	2	Maria Souza

Tabela Usuários

Row Key (id)	Usuario:nome
1	João Silva
2	Maria Souza

```
get 'Veiculos', 'ABC1234'

COLUMN                          CELL
Veiculo:cor                   timestamp=..., value=Azul
Usuario:id                    timestamp=..., value=1
Usuario:nome                  timestamp=..., value=Joao Silva
```

```
get 'Usuarios', '1'

COLUMN                        CELL
Usuario:nome                  timestamp=..., value=Joao Silva
```

Fonte: Autoria Própria (2025)

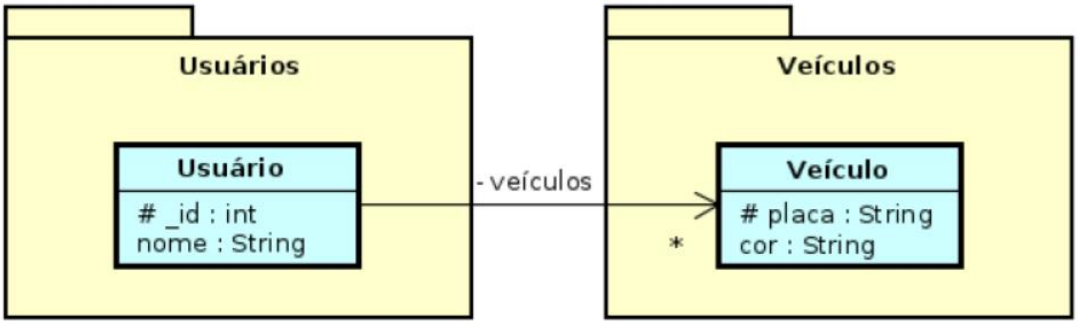
4.2.6. Duas Tabelas com *Array* de Elementos Referenciados

1- Descrição do Cenário Utilizando AML

Neste cenário, foi utilizado uma estrutura baseada em referências, onde a tabela Usuarios mantém um *array* de referências para os veículos que pertencem a cada usuário. Diferente dos modelos anteriores, aqui os veículos não são armazenados embutidos no usuário, mas sim em uma tabela separada Veiculos, permitindo que cada veículo seja acessado independentemente (figura 33). Essa abordagem traz um equilíbrio entre eficiência e flexibilidade, os usuários podem

acessar rapidamente as placas dos seus veículos sem precisar armazenar os dados completos dentro da linha do usuário, e os veículos são mantidos como registros independentes na tabela Veiculos, permitindo acesso individual eficiente sem redundância de informações. Tem-se também maior facilidade de atualização dos veículos, como os veículos não estão embutidos, modificações não impactam a estrutura do usuário. No entanto, esse modelo também apresenta desafios, especialmente em operações de leitura, que terão maior latência pois requer múltiplas consultas para recuperar todos os detalhes dos veículos de um usuário. Esse modelo é recomendado para cenários onde apenas os identificadores dos veículos precisam ser acessados frequentemente, caso todos os detalhes de cada veículos sejam necessários essa pode não ser a melhor abordagem.

Figura 33 – Cenário de Usuário com Um Array de Referências de Veículos



Fonte: MONTEIRO (2023)

Conforme ilustrado na figura 33, este cenário apresenta dois agregados, cada um composto por apenas uma entidade. O Agregado Usuários terá a entidade Usuário, identificada pelo campo _id e possuindo o atributo nome. O Agregado Veículos será composto pela entidade Veículo, identificada pela placa e possuindo o atributo cor. Por fim o relacionamento é estabelecido através do *link* de associação, indicando que um usuário pode ter vários veículos associados, porém sem repetições. A diferença para o primeiro cenário é que através do *link* de associação indica-se que a entidade Usuário irá armazenar um *array* de referências para Veículo ao invés de embutir todas as informações de Veículo.

2- Pontos Positivos

- **Menor Redundância de Dados:** Como não há embutimento no registro do usuário, evita-se duplicar as informações do veículo. Isso diminui o consumo de armazenamento e simplifica parte do processo de atualização.
- **Facilidade de Atualização:** Alterar dados de um veículo não exige reescrever a linha do usuário, pois o Veiculo está em uma tabela independente. Esse ponto reduz o overhead de escrita em cenários de modificações frequentes nos atributos dos veículos.
- **Flexibilidade de Consulta de Veículo:** A tabela de veículos pode ser consultada diretamente quando o contexto for “detalhes de um veículo”. Isso é especialmente útil quando a aplicação prioriza interações onde o veículo é a entidade principal.

3- Pontos Negativos

- **Múltiplas Leituras para Agrupar Detalhes**: Se a aplicação precisar recuperar todos os dados de um usuário e dos seus veículos, terá de realizar ao menos duas consultas (uma para obter o *array* de placas de veículos e outra(s) para recuperar seus detalhes na tabela de Veículos). Isso aumenta a latência em comparação a um modelo com embutimento.
- **Complexidade de Orquestração:** Embora menor que nos modelos de duplicação total, a aplicação ainda precisa gerenciar e sincronizar as referências do usuário para garantir que uma remoção ou inclusão de veículo seja refletida na lista de referências.

4- Impactos nas Características CAP do HBase

- **Consistência:** Cada linha de usuário ou de veículo permanece atomicamente consistente no HBase. Entretanto, manter a coerência entre o *array* de referências de um usuário e os registros efetivos de veículos (existência ou remoção) depende de processos na camada de aplicação, pois o HBase não impõe integridade referencial nativa.

- **Disponibilidade:** Caso a tabela de usuários esteja indisponível, não é possível recuperar facilmente as referências de veículos. De forma semelhante, se a tabela de veículos estiver indisponível, as consultas aos atributos completos dos veículos ficam inviabilizadas. O HBase (CP) pode sacrificar disponibilidade pontual para garantir a consistência por linha.
- **Particionamento:** Cada agregado é particionado pela sua chave primária, o que tende a distribuir a carga de forma mais uniforme, especialmente se o volume de veículos por usuário variar bastante. Entretanto, em caso de partição de rede, cada lado pode atualizar apenas parte das tabelas, resultando em possíveis inconsistências entre as referências do usuário e a tabela de veículos até a reconciliação pós-falha.

5- Exemplos de Tabelas e Consultas no HBase

Neste cenário existirão duas tabelas no HBase, a tabela Usuários, que terá os id’s como Row Key, essa tabela possuirá duas famílias de colunas. A família de colunas Usuário com a informação de nome, e a família de colunas Veículos, a qual terá em cada uma de suas colunas o identificador de um veículo (placa). A segunda tabela chamada Veículos, terá suas linhas identificadas pela placa, e apenas uma família de colunas (Veículo) com a informação da cor (figura 34).

Figura 34 – Exemplo de Tabelas e Consultas do Cenário 6

Tabela Usuários

Row Key (id)	Usuario:nome	Veiculos:1	Veiculos:2
1	João Silva	ABC1234	DEF5678
2	Maria Souza	GHI9012	JKL3456

Tabela Veículos

Row Key (placa)	Veiculo:cor
ABC1234	Azul
DEF5678	Preto
GHI9012	Vermelho
JKL3456	Branco

```
get 'Usuarios', '1'

COLUMN                          CELL
Usuario:nome                    timestamp=..., value=Joao Silva
Veiculos:1                      timestamp=..., value=ABC1234
Veiculos:2                      timestamp=..., value=DEF5678
```

```
get 'Veiculos', 'ABC1234'

COLUMN                            CELL
Veiculo:cor                       timestamp=..., value=Azul
```

Fonte: Autoria Própria (2025)

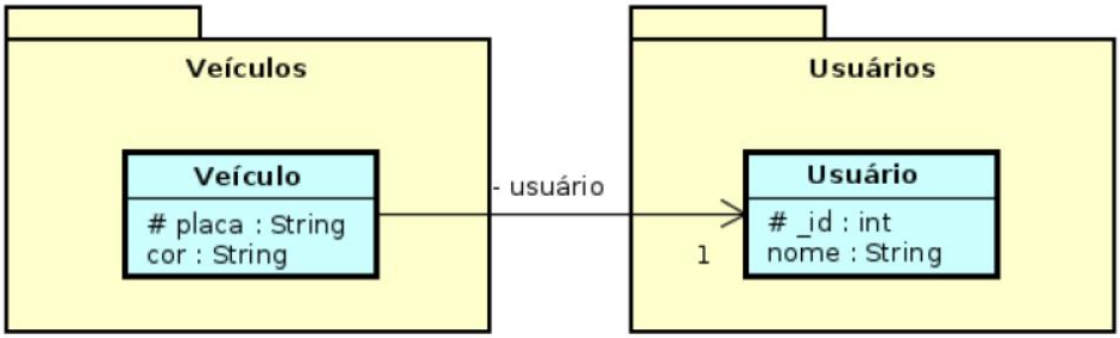
4.2.7. Duas Tabelas com Apenas um Elemento Referenciado

1- Descrição do Cenário Utilizando AML

Este cenário é de certa forma similar ao cenário anterior, cada veículo armazena apenas uma referência ao seu usuário, sem embutir seus dados completos, conforme ilustrado na figura 35. Isso significa que, ao contrário de modelos anteriores onde os veículos armazenavam o nome do usuário embutido, aqui tem-se duas tabelas separadas (Usuarios e Veiculos), onde a tabela Veiculos contém apenas um identificador do usuário. Essa abordagem é totalmente baseada em referências, eliminando qualquer redundância de informações entre as tabelas.

Cada usuário é armazenado apenas uma vez na tabela Usuarios, e cada veículo contém apenas o identificador do usuário ao qual pertence. Para recuperar os detalhes completos de um veículo e de seu respectivo dono, é necessário realizar uma consulta na tabela Veiculos seguida por uma segunda consulta na tabela Usuarios para obter os dados do usuário. A vantagem desse modelo está na eficiência da escrita e na consistência dos dados, pois evita duplicação de informações e facilita atualizações. No entanto, essa abordagem aumenta a complexidade das consultas, pois qualquer busca que necessite exibir os veículos juntamente com os dados do usuário exigirá múltiplas consultas. Essa estratégia é recomendada para sistemas onde a consistência dos dados é prioritária e onde os veículos são acessados mais frequentemente de forma individual. No entanto, se a aplicação exigir consultas frequentes envolvendo tanto usuários quanto veículos simultaneamente, pode ser necessário considerar estratégias híbridas para otimizar o desempenho.

Figura 35 – Cenário de Veículos com Referência para Um Usuário



Fonte: MONTEIRO (2023)

Neste cenário existirão dois agregados, cada um composto por apenas uma entidade. O Agregado Veículos será composto pela entidade Veículo, identificada pela placa e possuindo o atributo cor. O Agregado Usuários terá a entidade Usuário, identificada pelo campo _id e possuindo o atributo nome. Por fim o relacionamento é estabelecido através do *link* de associação, indicando que um veículo pode ter um usuário associados (figura 35). A diferença para o cenário anterior é que a entidade Veículo irá armazenar uma referência para Usuário.

2- Pontos Positivos

- **Menor Redundância de Dados:** Como não há embutimento no registro do veículo, evita-se duplicar as informações do usuário. Isso diminui o consumo de armazenamento e simplifica parte do processo de atualização.
- **Facilidade de Atualização:** Alterar dados de um usuário não exige reescritas de múltiplas linhas; basta atualizar o registro do usuário na tabela correspondente. Da mesma forma, atualizar um veículo não afeta os registros de usuários.
- **Flexibilidade de Consulta de Usuário:** A tabela de usuários pode ser consultada diretamente quando o contexto for “detalhes de um usuário”. Isso é especialmente útil quando a aplicação prioriza interações onde o usuário é a entidade principal.

3- Pontos Negativos

- **Múltiplas Leituras para Agrupar Detalhes**: Sempre que for preciso exibir simultaneamente dados do usuário e do veículo, será necessário efetuar pelo menos duas consultas, uma em cada tabela, aumentando a latência.
- **Falta de Agrupamento:** Não existe uma forma direta de recuperar todos os veículos de um usuário sem realizar varreduras ou indexar a relação de outra maneira, pois o usuário não mantém um array de veículos nem embute as informações do veículo.
- **Controle de Integridade Referencial**: O HBase não impõe *constraints* entre tabelas. Dessa forma, a aplicação precisa garantir que um Id de usuário existente em Veículos realmente corresponda a uma linha válida na tabela Usuários.

4- Impactos nas Características CAP do HBase

- **Consistência:** Cada linha seja da tabela Usuários ou Veículos tem consistência forte por conta do modelo de escrita do HBase, porém a integridade de referência recai completamente na lógica de aplicação.

- **Disponibilidade:** Caso uma das tabelas fique indisponível ainda é possível consultar a outra. Contudo, para obter dados conjuntos, é preciso que ambas estejam operacionais.
- **Particionamento:** Cada tabela é particionada independentemente pela chave de linha. Em caso de partição de rede, cada lado pode continuar operando, mas resulta em possíveis atrasos ou falhas ao correlacionar um veículo ao seu usuário, até que a conexão se normalize.

5- Exemplos de Tabelas e Consultas no HBase

Neste cenário existirão duas tabelas no HBase, a tabela Veículos, que terá as placas como Row Key, essa tabela possuirá duas famílias de colunas. A família de colunas Veículo com a informação de cor, e a família de colunas Usuário, a qual terá como valor o id de um usuário. A segunda tabela chamada Usuários, terá suas linhas identificadas pelo id do usuário, e apenas uma família de colunas (Usuário) com a informação do seu nome (figura 36).

Figura 36 – Exemplo de Tabelas e Consultas do Cenário 7

Tabela Veículos

Row Key (placa)	Veiculo:cor	Usuario:id
ABC1234	Azul	1
DEF5678	Preto	1
GHI9012	Vermelho	2
JKL3456	Branco	2

Tabela Usuários

Row Key (id)	Usuario:nome
1	João Silva
2	Maria Souza

```
get 'Veiculos', 'ABC1234'

COLUMN                          CELL
Veiculo:cor                     timestamp=..., value=Azul
Usuario:id                      timestamp=..., value=1
```

```
get 'Usuarios', '1'

COLUMN                        CELL
Usuario:nome                  timestamp=..., value=Joao Silva
```

```
scan 'Veiculos', {FILTER => "SingleColumnValueFilter('Usuario', 'id', =, 'binary:1')"}

ROW                           COLUMN                        CELL
ABC1234                       Veiculo:cor                   value=Azul
.                             Usuario:id                    value=1
DEF5678                       Veiculo:cor                   value=Preto
.                             Usuario:id                    value=1
```

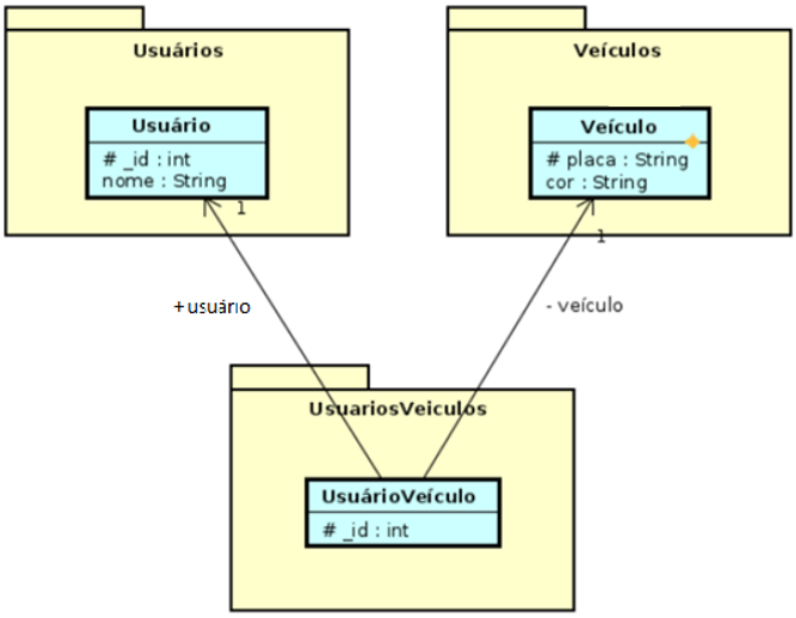
Fonte: Autoria Própria (2025)

4.2.8. Três Tabelas Sendo uma Intermediária de Referências

1- Descrição do Cenário Utilizando AML

Neste último cenário, foi adotado uma abordagem totalmente baseada em referências, utilizando três tabelas para armazenar separadamente as entidades e suas relações. Aqui, os usuários e veículos são armazenados em tabelas distintas, enquanto uma terceira tabela intermediária é utilizada para representar a relação entre eles, conforme ilustrado na figura 37. Dessa forma, elimina-se qualquer redundância, garantindo que cada entidade seja armazenada apenas uma vez e que as referências sejam mantidas separadamente. Esse modelo é altamente escalável e flexível, pois permite consultas eficientes tanto por usuários quanto por veículos sem a necessidade de duplicação de informações. No entanto, essa abordagem também aumenta a complexidade da leitura, pois para obter todas as informações de um usuário e seus veículos, são necessárias múltiplas consultas. Esse modelo é recomendado para sistemas onde os veículos e os usuários são frequentemente acessados de forma independente, e onde a escalabilidade e a integridade dos dados são mais importantes do que a velocidade das consultas combinadas.

Figura 37 – Cenário com Terceira Tabela de Referências



Fonte: MONTEIRO (2023)

No cenário da figura 37 são evidenciados três agregados, cada um composto por apenas uma entidade. O Agregado Usuários terá a entidade Usuário, identificada pelo campo _id e possuindo o atributo nome. O Agregado Veículos será composto pela entidade Veículo, identificada pela placa e possuindo o atributo cor. Por fim o agregado UsuáriosVeículos apenas com o atributo identificador _id. Este terceiro agregado representa o relacionamento de posse dos veículos, indicando através de um link de associação para Usuário com o pictograma regular, que o _id do usuário pode aparecer várias vezes na tabela de relacionamento. Também tem-se o *link* de associação para Veículo com o pictograma *unique*, indicando que a placa do veículo pode aparecer apenas uma vez.

2- Pontos Positivos

- **Eliminação de Redundância:** Cada entidade é armazenada apenas uma vez, o que reduz o consumo de espaço e facilita a manutenção dos dados.
- **Escalabilidade Flexível:** É possível escalar cada tabela de forma independente, distribuindo usuários e veículos em diferentes regiões, sem sobrecarregar as linhas com dados embutidos ou duplicados.
- **Alto Grau de Normalização:** A aplicação “limpa” as interdependências de dados, uma vez que cada relação é explicitamente gerenciada na tabela intermediária, auxiliando na clareza de regras de negócio que exijam relacionamentos complexos.

3- Pontos Negativos

- **Consultas Compostas mais Custosas:** Para reunir todos os veículos de um usuário, ou vice-versa, são necessárias pelo menos duas consultas (na tabela de relacionamento para recuperar as chaves e, em seguida, na tabela de destino). Isso pode impactar a latência em leituras que precisem unir dados de múltiplas entidades.
- **Complexidade de Gerenciamento:** Manter três tabelas distintas exige um controle adicional na camada de aplicação, assegurando a criação e remoção

de relacionamentos em paralelo à inserção ou exclusão de usuários e veículos.

- **Sem Integridade Referencial Nativa:** Assim como nos outros cenários com referências, o HBase não impõe *constraints*. A responsabilidade de manter a coerência (por exemplo, não deixar relacionamentos órfãos) recai totalmente na lógica de negócios.

4- Impactos nas Características CAP do HBase

- **Consistência:** Cada linha permanece consistentemente atualizada no nível de linha do HBase, mas a integridade entre as três tabelas requer coordenação externa. Se ocorrer falha na atualização da tabela de relacionamento, pode haver discrepância entre os dados de usuários e veículos.
- **Disponibilidade:** Caso uma das tabelas fique indisponível, é possível consultar as outras duas. Porém, não será viável montar relacionamentos completos se a tabela de relacionamento ou a tabela buscada estiver offline. Em sistemas “CP” como o HBase, a disponibilidade pode ser momentaneamente afetada em caso de falha do *RegionServer* responsável por cada tabela.
- **Particionamento:** As três tabelas podem ser particionadas com base em chaves, permitindo escalabilidade horizontal. Em caso de partição de rede, cada lado pode atualizar apenas parte das tabelas, dificultando a sincronização dos relacionamentos até o restabelecimento.

5- Exemplos de Tabelas e Consultas no HBase

Neste cenário existirão três tabelas no HBase, a tabela Veículos, que terá as placas como Row Key, essa tabela possuirá uma família de colunas, a família de colunas Veículo com a informação de cor. A segunda tabela chamada Usuários, terá suas linhas identificadas pelo id do usuário, e apenas uma família de colunas (Usuário) com a informação do seu nome. E uma terceira tabela chamada de UsuárioVeículos que terá um _id como Row Key e duas famílias de colunas, Usuário

e Veículo, cada uma com apenas um valor, suas respectivas referências de chaves para suas tabelas de destino (figura 38).

Figura 38 – Exemplo de Tabelas e Consultas do Cenário 8

Tabela Veículos

Row Key (placa)	Veiculo:cor
ABC1234	Azul
DEF5678	Preto
GHI9012	Vermelho
JKL3456	Branco

Tabela Usuários

Row Key (id)	Usuario:nome
1	João Silva
2	Maria Souza

Tabela UsuárioVeículos

Row Key (id)	Usuario:id	Veiculo:placa
1	1	ABC1234
2	1	DEF5678
3	2	GHI9012
4	2	JKL3456

```
get 'Veiculos', 'ABC1234'

COLUMN                          CELL
Veiculo:cor                     timestamp=..., value=Azul
```

```
get 'Usuarios', '1'

COLUMN                          CELL
Usuario:nome                    timestamp=..., value=Joao Silva
```

```
scan 'UsuarioVeiculos', {FILTER => "SingleColumnValueFilter('Usuario', 'id', =, 'binary:1')"}

ROW                             COLUMN                          CELL
1                               Veiculo:placa                   value=ABC1234
2                               Veiculo:placa                   value=DEF5678
```

Fonte: Autoria Própria (2025)

4.3. RESULTADOS E CONSIDERAÇÕES FINAIS

Os cenários descritos apresentam diferentes estratégias de modelagem, cada qual projetada para atender a prioridades específicas, como velocidade de acesso, facilidade de manutenção ou redução de redundâncias. Ao longo desta seção, foram detalhadas oito variações de modelagem lógica para um mesmo cenário, considerando múltiplas possibilidades de composição e associação, diferentes graus de embutimento e referência, além de níveis variados de redundância. Reconhece-se que não há um modelo universal que sirva a todos os contextos, mas sim uma série de alternativas para equilibrar exigências de consistência, disponibilidade e desempenho. A seguir, apresenta-se uma tabela-síntese consolidando as principais vantagens e desvantagens de cada cenário.

Tabela 2 - Síntese das Avaliações

Cenário	Abordagem	Vantagem Principal	Desvantagem Principal
1	**Composição** (Usuário → Veículos embutidos); sem redundância	Alta eficiência ao recuperar todos os veículos de um usuário em uma única leitura	Veículos não podem ser acessados isoladamente
2	**Composição + Tabela Redundante** (Usuário → Veículos embutidos + Veículos em tabela separada); redundância parcial	Combina leituras rápidas via embutimento com acesso independente ao veículo	Sincronização complexa devido à duplicação parcial
3	**Composição + Redundância Total** (duplicação total: tanto usuário quanto veículo repetidos)	Consultas extremamente rápidas em ambos os contextos (do usuário ou do veículo)	Consumo de armazenamento elevado e maior custo de manutenção
4	**Composição** (Veículo → Usuário embutido); sem redundância	Focado em leituras centradas no veículo, trazendo dados do usuário em uma única linha	Recuperar todos os veículos de um usuário exige varredura da tabela de veículos
5	**Composição + Tabela Redundante** (Veículo → Usuário embutido + Usuário em tabela separada); redundância parcial	Leituras eficientes tanto pela placa do veículo quanto diretamente pelos dados do usuário	Exige atualizações sincronizadas em duas tabelas, duplicando esforços de escrita
6	**Associação** (Usuário mantém *array* de referências a Veículos); sem redundância	Atualizar dados do veículo é mais simples, pois ele está isolado em tabela própria	Montar informações completas (usuário + detalhes do veículo) requer múltiplas consultas
7	**Associação** (Veículo mantém apenas referência a Usuário); sem redundância	Baixa redundância e escalabilidade independente para cada tabela	A busca combinada (veículo + dados do usuário) sempre exige ao menos duas operações de leitura
8	**Três Tabelas (Intermediária de Referências)** (Usuário, Veículo e Relacionamento)	Alto grau de normalização, eliminando duplicações	Demanda múltiplas consultas para agregar dados (usuário + veículo) e maior complexidade de lógica

Fonte: Autoria Própria (2025)

Com base na análise realizada, é possível propor algumas diretrizes práticas para orientar a escolha entre os cenários apresentados, considerando o tipo de aplicação, o padrão de acesso aos dados e restrições arquiteturais:

- **Cenários 1 e 4 (composição sem redundância):** recomendados para acessos centrados no agregado, quando se deseja obter todos os dados relacionados em uma única leitura (ex: recuperar o usuário com todos os veículos, ou o veículo com os dados do usuário).
- **Cenários 2, 3 e 5 (composição com redundância):** mais adequados em sistemas que exigem flexibilidade de leitura em diferentes perspectivas, como painéis de administração ou APIs REST que precisam acessar usuários e veículos de forma independente. A escolha entre parcial (2, 5) ou total (3) depende do volume de dados e da frequência de atualização.
- **Cenários 6 e 7 (referência com baixa redundância):** recomendados para aplicações com alta taxa de escrita e atualização isolada de entidades, como sistemas transacionais ou serviços com atualizações frequentes por parte dos usuários.
- **Cenário 8 (tabela intermediária):** indicado em situações de relacionamento complexo ou N:N, como quando um mesmo veículo pode estar associado a múltiplos usuários (ex: caronas, frota corporativa), ou quando há necessidade de normalização máxima para evitar redundância.

Em todos os casos, é importante considerar a frequência de leitura e escrita, a criticidade da consistência entre entidades e a estrutura organizacional dos dados na aplicação. Não existe um cenário universalmente melhor, mas sim escolhas otimizadas para contextos específicos.

Além de comparar os impactos teóricos de cada estratégia, a aplicação da AML nos diversos cenários permitiu demonstrar, de forma sistemática, a completude da linguagem ao mapear diretamente os principais elementos do modelo colunar do HBase, como *rowkeys*, famílias de colunas, atributos compostos e estruturas referenciadas. Cada construtor utilizado nos diagramas AML foi capaz de expressar adequadamente os aspectos essenciais da organização lógica do HBase, evidenciando que a linguagem possui sintaxe, semântica e abstrações compatíveis

com os conceitos nativos desse tipo de banco de dados. Contudo, também se observou que a AML vai além dos requisitos estruturais demandados pelo HBase, apresentando uma expressividade superior por meio de seus construtores. Essa expressividade ampliada permite modelagens mais ricas, mas também introduz restrições formais adicionais, derivadas da semântica estática da linguagem. A AML impõe critérios de formação que não são necessariamente exigidos pelo HBase, mas que precisam ser respeitados no momento da implementação, logo a camada de aplicação deve embarcar tais restrições. Embora a AML se mostre completa no que diz respeito à representação do modelo colunar do HBase, ela também atua como uma camada adicional de controle e robustez na modelagem, promovendo uma representação mais precisa e segura dos dados. Essa característica fortalece sua aplicabilidade em sistemas distribuídos e escaláveis, especialmente em contextos nos quais a modelagem lógica precisa ser validada formalmente antes de ser transposta para o ambiente físico.

5. CONCLUSÃO

A modelagem de bancos de dados NoSQL, especificamente em bancos de família de colunas, exige abordagens diferenciadas em relação aos modelos tradicionais relacionais. Neste trabalho, foi explorada a *Aggregate Modeling Language* como uma ferramenta de modelagem lógica, aplicando seus construtores para representar diferentes cenários de organização e estruturação de dados no HBase. A AML demonstrou-se uma linguagem eficaz para estruturar agregados de dados, facilitando a transição de conceitos mais amplos, como *Domain-Driven Design*, para a realidade de bancos distribuídos e escaláveis. Ao longo do estudo, diferentes estratégias de modelagem foram analisadas, avaliando suas implicações em desempenho, consistência e flexibilidade de acesso aos dados. Cada abordagem apresentou vantagens e desvantagens, sendo possível identificar *trade-offs* entre eficiência na leitura, latência na escrita e complexidade de manutenção. Modelos que embutem dados proporcionam consultas rápidas e maior localidade de referência, mas podem impactar a flexibilidade e escalabilidade do sistema. Por outro lado, abordagens baseadas em referências e tabelas intermediárias permitem maior normalização dos dados, reduzindo redundância, mas ao custo de maior latência nas consultas complexas.

A partir dos exemplos discutidos no Capítulo 4, este trabalho pode servir como referência prática para modelagem de bancos de família de colunas, auxiliando arquitetos e desenvolvedores a tomar decisões informadas ao projetar suas estruturas de dados. O estudo demonstrou como diferentes padrões de modelagem podem impactar a organização e o acesso aos dados, oferecendo um guia prático para avaliar a melhor abordagem dependendo dos requisitos da aplicação. Contudo, é importante destacar que as soluções apresentadas foram aplicadas ao HBase, podendo haver variações em outras implementações de bancos de dados NoSQL de família de colunas. Além disso, novas demandas e evoluções tecnológicas podem introduzir novas práticas de modelagem, reforçando a necessidade de atualizações contínuas no conhecimento sobre modelagem de dados distribuídos.

Dessa forma, este trabalho contribui para a compreensão da modelagem de bancos de família de colunas utilizando AML, demonstrando sua completude como linguagem de modelagem lógica capaz de representar com fidelidade os principais

elementos do Apache HBase. Ao promover um entendimento mais aprofundado das implicações arquiteturais envolvidas e oferecer um mapeamento direto entre os construtores da AML e as estruturas do modelo colunar, o estudo amplia as possibilidades de aplicação dessa linguagem em projetos que exigem escalabilidade, consistência e clareza na modelagem. A crescente adoção de bancos NoSQL torna fundamental o domínio dessas técnicas, garantindo que sistemas modernos sejam eficientes, escaláveis e alinhados às necessidades das aplicações distribuídas contemporâneas.

6. TRABALHOS FUTUROS

Os cenários analisados neste trabalho concentraram-se em relacionamentos do tipo 1:N, refletindo a estrutura conceitual adotada como base. No entanto, em domínios com interações mais complexas entre entidades, é comum a ocorrência de relacionamentos do tipo N:N, os quais demandam estratégias específicas de modelagem. Como desdobramento natural, propõe-se a ampliação da abordagem para incluir cenários baseados em relacionamentos N:N, explorando alternativas como tabelas intermediárias, duplicação seletiva de dados e variações nas formas de referência e composição entre entidades. A aplicação da AML nesses novos contextos permitirá avaliar sua expressividade e adequação para modelagens ainda mais complexas.

Além disso, embora este trabalho tenha discutido os impactos teóricos de cada estratégia de modelagem com base em critérios qualitativos como escalabilidade, manutenção e eficiência de acesso, se faz importante validar essas análises por meio de experimentos empíricos baseados em benchmarks de desempenho. A execução de testes sistemáticos, utilizando ferramentas como YCSB (*Yahoo! Cloud Serving Benchmark*), permitirá mensurar métricas objetivas, como latência de leitura e escrita, *throughput*, uso de armazenamento e resiliência a falhas, sob diferentes cargas e perfis de acesso. Esses resultados não apenas reforçariam as conclusões teóricas aqui apresentadas, mas também possibilitariam a formulação de heurísticas práticas de escolha de estratégias de modelagem, com base em parâmetros observáveis do sistema. Em conjunto, a extensão para relacionamentos N:N e a condução de benchmarks experimentais configuram os próximos passos naturais desta pesquisa, ampliando sua relevância prática e contribuindo para a consolidação da AML como linguagem de modelagem lógica eficaz em ambientes NoSQL colunares.

REFERÊNCIAS

[1] **ADYA, A.; GRANDL, R.; MYERS, D. S.; QIN, H.** Fast key-value stores. *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019. Disponível em: https://doi.org/10.1145/3317550.3321434. Acesso em: 02 jan. 2025.

[2] **AL-GHIFARI, Muhammad Fauzan; AZIZAH, Fazat Nur.** Development of application for entity-relationship diagram conversion to logical schema of NoSQL column-oriented. In: *International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 2022. p. 30-35.

[3] **DA SILVA DIAS, Ariel.** *Bancos de dados não relacionais.* São Paulo: Editora Senac, 2023.

[4] **DORNELLES, Carlos Alberto.** Projeto de Banco de Dados. Disponível em: https://www.cadcobol.com.br/db2_novo_projeto_banco_dados.htm. Acesso em: 02 jan. 2025.

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

[6] **FROZZA, Angelo Augusto; SCHREINER, Geomar André; DOS SANTOS, Ronaldo.** Projeto de Bancos de Dados NoSQL. In: *Tópicos em Gerenciamento de Dados e Informações.* Anais do Simpósio Brasileiro de Banco de Dados, 2022.

[7] **GARCIA, Vinícius Salles; SOTTO, Eder Carlos Salazar.** Comparativo entre os modelos de banco de dados relacional e não-relacional. *Revista Interface Tecnológica*, v. 16, n. 2, p. 12-24, 2019.

[8] **GÓMEZ, Paola; RONCANCIO, Claudia; CASALLAS, Rubby.** Analysis and evaluation of document-oriented structures. Data & Knowledge Engineering, v. 134, p. 101893, 2021.

[9] **HADDADI, Saad; OULAHYANE, Kaoutar; OUFTTOU, Mohamed.** An introduction to Apache HBase. *Hands On Apache HBase*, 28 maio 2020. Disponível em: https://medium.com/hands-on-apache-hbase/an-introduction-to-apache-hbase-2cdd1d9ff13. Acesso em: 25 fev. 2025.

[10] **HASSAN, Muhammad Umair et al.** A comprehensive study of HBase storage architecture—a systematic literature review. *Symmetry*, v. 13, n. 1, p. 109, 2021.

[11] **KAUFMANN, Michael; MEIER, Andreas.** *SQL and NoSQL Databases.* Switzerland: Springer, 2023.

[12] **LIMA, C.; MELLO, R.** A Workload-driven Logical Design Approach for NoSQL Document Databases. In: *17th ACM International Conference on Information Integration and Web-based Applications & Services (iiWAS)*. ACM, 2015. p. 73:1–73:10.

[13] **LIMA, Gênesis Jeferson Ferreira Pereira de**. Uma linguagem de modelagem para refatoração estrutural de banco de dados relacionais. 2018. Dissertação de Mestrado. Universidade Federal de Pernambuco.

[14] **MA, R.; ZHOU, W.; MA, Z.** An efficient NoSQL-based storage schema for large-scale time series data. *Journal of Database Management*, v. 35, n. 1, p. 1-21, 2024. Disponível em: https://doi.org/10.4018/jdm.339915. Acesso em: 02 jan. 2025.

[15] **MICROSOFT LEARN**. *Migrar dados do Apache HBase para uma conta do Azure Cosmos DB for NoSQL*. 2024. Disponível em: https://learn.microsoft.com/pt-br/azure/cosmos-db/nosql/migrate-hbase-to-cosmos-db Acesso em: 13 mar. 2025.

[16] **MONTEIRO, Thomas Anderson Feitosa.** *Modelagem de banco de dados orientados a documentos com AML.* 2023. Trabalho de Conclusão de Curso (Graduação) – Universidade Federal de Pernambuco. Disponível em: https://repositorio.ufpe.br/handle/123456789/52945. Acesso em: 18 dez. 2024.

[17] **POFFO, João Paulo; MELLO, R. dos S.** A Logical Design Process for Columnar Databases. In: *Eleventh International Conference on Internet and Web Applications and Services.* 2016.

[18] **SADALAGE, P. J.; FOWLER, M.** *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley, 2013.

[19] **SCHREINER, G.; DUARTE, D.; MELLO, R.** Bringing SQL Databases to Key-based NoSQL Databases: A Canonical Approach. *Computing*, v. 102, n. 1, p. 221–246, 2020.

[20] **SCHREINER, G. A.; DUARTE, D.; MELLO, R.** SQLtoKey-NoSQL: A Layer for Relational to Key-based NoSQL Database Mapping. In: *17th ACM International Conference on Information Integration and Web-based Applications & Services (iiWAS)*. ACM, 2015. p. 74:1–74:9.

[21] **SHETH, Vinay.** *Comparative Performance Analysis of Column Family Databases: Cassandra and HBase.* 2023. Tese (Doutorado) – Dhirubhai Ambani Institute of Information and Communication Technology.

[22] **SHAHIDA, B.** Examining and juxtaposing the two NoSQL databases MongoDB and CouchDB. *International Journal of Research Publication and Reviews*, p. 884-888, 2022. Disponível em: https://doi.org/10.55248/gengpi.2022.3.9.28. Acesso em: 02 jan. 2025.

[23] **SOARES, Bruno Eduardo; BOSCARIOLI, Clodis.** Modelo de Banco de Dados Colunar: Características, Aplicações e Exemplos de Sistemas. In: *Escola Regional de Banco de Dados* (IX ERBD–SBC). Sociedade Brasileira de Computação, Camobiú, 2013.

[24] **TSAI, Chia-Ping et al.** The Time Machine in Columnar NoSQL Databases: The Case of Apache HBase. *Future Internet*, v. 14, n. 3, p. 92, 2022.

[25] **VENKATRAMAN, Sitalakshmi et al.** SQL versus NoSQL movement with big data analytics. *International Journal of Information Technology and Computer Science*, v. 8, n. 12, p. 59-66, 2016.