Revisão sistemática sobre a camada de acesso em banco de dados poliglotas

Ana Paula Hartmann[1], **Denio Duarte** [1], **Geomar André Schreiner**[1]

[1] Universidade Federal da Fronteira Sul (UFFS)
SC-484, Km 02 - Fronteira Sul, Chapecó – SC, 89815-899

anahartmann.paula@gmail.com, {duarte,gschreiner}@uffs.edu.br

Abstract. *Polyglot databases, also known as polystores, emerged to address the requirements of Big Data for managing multiple databases and integrating queries across diverse data models. Within the various types of polystores, access-layer solutions are responsible for query decomposition, optimization, translation into system-specific languages, and result integration. Consequently, this study presents a systematic literature review of access-layer solutions published over the last decade (2015–2025). Based on predefined selection criteria, six studies were identified. The findings demonstrate that the majority of these solutions support the primary NoSQL data models and the relational model. Furthermore, the supported query languages are predominantly SQL-like, and nearly all implementations utilize schemas.*

Resumo. *Banco de dados poliglotas, ou polystores, nasceram com o objetivo de atender às necessidades impostas pelo Big Data para gerenciar múltiplos bancos e integrar as consultas de diversos modelos de dados. Dentro dos tipos de polystores, os de camadas de acesso são os responsáveis decompor consultas, otimizar, traduzir para as linguagens dos sistemas e integrar os resultados. Este trabalho, então, apresenta uma revisão sistemática da literatura sobre as soluções de camadas de acesso publicadas na última década (2015–2025). A partir de conjunto de critérios de seleção, foram elencados seis trabalhos e os resultados mostram que que a maior parte suporta os principais modelos de dados NoSQL, assim como o modelo relacional. As linguagens de consulta aceitas são em sua maioria SQL-Like e quase todos utilizam esquemas.*

1. Introdução

Os Banco de Dados Relacionais (BDR) são, ainda nos dias de hoje, o modelo mais utilizado por aplicações centradas em dados [DB-Engines 2026]. Porém, segundo [Sadalage and Fowler 2013], por conta da incompatibilidade de impedância, termo que descreve a incompatibilidade entre o modelo relacional e as estruturas de dados em memória das linguagens de programação, houveram adversidades com o limite que os BDR impõem. Essa diferença obriga o desenvolvedor a realizar uma tradução constante entre o modelo orientado a objetos e o modelo relacional, gerando uma complexidade adicional ao processo. Essas limitações culminaram com o surgimento dos bancos de dados (BDs) conhecidos como NoSQL, que não seguem as mesmas regras dos relacionais e possuem características como a alta disponibilidade e alta escalabilidade essenciais em ambientes de *BigData*.

Entretanto, tanto os BDR quanto os NoSQL possuem características que os favorecem para tipos de aplicações específicas. Dados com alta conectividade, como estruturas de redes sociais, são naturalmente representados por modelos de grafos, otimizando a travessia de adjacências em bancos NoSQL orientados a grafos. Em contrapartida, a representação desses dados em modelos relacionais impõe uma sobrecarga operacional devido ao elevado número de junções (*joins*), o que compromete a escalabilidade e aumenta a complexidade da manutenção. Diante desse cenário, as organizações frequentemente adotam a persistência poliglota para maximizar o desempenho. Essa diversidade pode causar problemas de gerenciamento dos dados devido às várias formas de acesso, dependendo do modelo. Para suprir essa necessidade, BDs poliglotas, ou polystores, foram propostos com o objetivo de gerenciar modelos de dados heterogêneos e permitir o acesso ao dados através de uma interface de consultas unificada [Duggan et al. 2015].

Os polystores podem ser construídos de duas formas principais: como sistemas nativos ou como uma camada de acesso (*middleware*). Um polystore *middleware* (e.g. [Duggan et al. 2015]) atua principalmente como uma camada de acesso que se sobrepõe a BDs existentes. Eles são responsáveis por decompor, otimizar e traduzir consultas para as linguagens nativas dos sistemas e integrar os resultados retornados. Um polystore nativo (e.g. [Karimov et al. 2019]) é construído como um sistema unificado que mantém seus próprios componentes de armazenamento com diferentes modelos. Essa arquitetura permite que o polystore migrar, replicar ou particionar dados automaticamente nesses componentes, sem a intervenção do usuário.

Esse trabalho tem por objetivo fazer uma revisão sistemática sobre as publicações mais relevantes (*e.g.*, veículos de publicação e número de citações) sobre camadas de acesso de bancos poliglotas. Usando estes critérios foram selecionados seis trabalhos de 2015 a 2025. A revisão mostrou que a maior parte dos trabalhos pesquisados suporta os principais modelos de dados NoSQL (*i.e.*, grafo, documentos, chave-valor e família de colunas), assim como o modelo relacional. As linguagens de consulta aceitas são em sua maioria *SQL-Like*, ou o próprio SQL. Sobre a estrutura, quase todos os trabalhos consideram algum tipo de esquema, sendo mais frequente o global do que o local.

As próximas sessões são divididas da seguinte forma: Seção 2 apresenta alguns trabalhos relacionados; aspectos sobre banco de dado poliglota são brevemente apresentados na Seção 3; Seção 4 apresenta os trabalhos analisados; Seção 5 discute sobre os resultados de cada um e por fim a seção 6 conclui a revisão sistemática.

2. Trabalhos Relacionados

Na literatura, é possível encontrar trabalhos que façam uma revisão sistemática sobre polystores, [Araujo et al. 2025] [Silva et al. 2024], ou um *benchmark* [Karimov et al. 2019], porém, não focados na camada de acesso.

Em [Araujo et al. 2025] foi realizada uma revisão sobre à evolução de esquemas em aplicações de persistência poliglota. Esse termo se refere a capacidade de uma aplicação de utilizar diferentes modelos de BDs para funções específicas. Os autores estudam os fatores que levam a evolução dos esquemas, como são projetadas as arquiteturas de esquemas que podem evoluir e quais foram os modelos base para cada estudo. Já [Silva et al. 2024] trata sobre a modelagem conceitual de aplicações poliglotas. As perguntas utilizadas para a revisão sistemática abrangem os modelos lógicos de dados

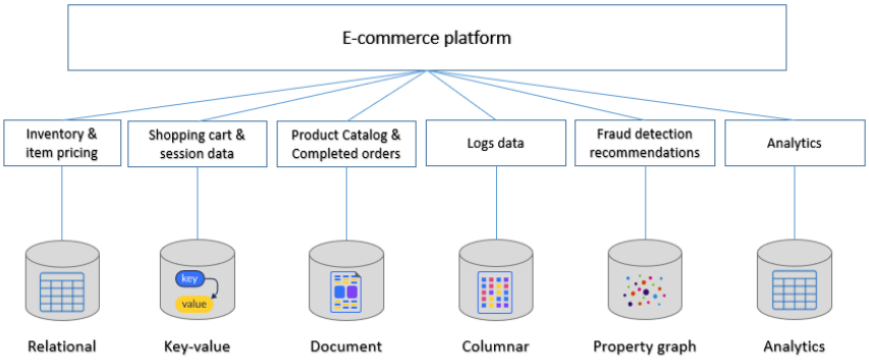

Figura 1. Exemplo de uma aplicação polystore[1].

considerados pela abordagem dos trabalhos, os requisitos propostos e os processos desenvolvidos. Por fim, [Karimov et al. 2019] tem por objetivo fazer uma avaliação experimental de polystores usando métricas como tempo total de execução, latência, tempo de execução individual, latência individual, tempo ocioso e escalabilidade.

O presente trabalho difere dos apresentados por fazer uma revisão sistemática focada em sistemas poliglotas que apresentem uma camada de acesso, distinguindo-se dos polystores nativos. Além disso, este trabalho se propõe a realizar uma comparação teórica de características sem a intenção de realizar experimentos com as ferramentas.

3. Banco de Dados Poliglotas

Os BDs federados começaram a ser estudados e implementados nas décadas de 1980 e 1990. A motivação era integrar sistemas já existentes preservando a autonomia de cada BD [Sheth and Larson 1990]. Porém, BDs federados tentavam propor um esquema para todas as bases de dados do sistema e visavam a integração de diferentes bases de dados. Essas características não atendem mais os requisitos das aplicações atuais, em que além de um acesso integrado o foco está em armazenar as informações em diferentes modelos de dados para obter maior desempenho [Stonebraker and Cetintemel 2005].

Para atender os requisitos das aplicações mais modernas, *i.e.*, BDs e esquemas heterogêneos, nasceram os sistemas poliglotas. O nome Banco de Dados Poliglotas, ou Polystores, surge com o trabalho de [Duggan et al. 2015], que apresenta o sistema *BigDAWG*. Os autores definem que Polystore é um sistema de gerenciamento de dados que integra múltiplos mecanismos de armazenamento baseados em diferentes modelos de dados e linguagens de consulta. Isso permite o acesso unificado e a utilização do melhor modelo para cada tipo de informação. A Figura 1 apresenta um exemplo de uma aplicação (*e-commerce*) que se apoia em outras aplicações acessando diferentes modelos de banco de dados.

A partir desse marco na pesquisa voltada para BDs, foram desenvolvidos diversos outros polystores. Alguns exemplos de trabalhos são: [Vogt et al. 2018] e [Podkorytov and Gubanov 2019] que constroem um sistema de gerenciamento de BDs polystore implementado como um único sistema integrado. Outros trabalham em melhorar a arquitetura do *BigDAWG*, como o [Yu et al. 2019] que faz uma movimentação de

[1]hackolade.com/polyglot-data-modeling.html

dados baseada em RDMA para melhorar pontos do trabalho original.

Um polystore pode ser um sistemas nativos ou uma camada de acesso (*middleware*). Um polystore nativo é projetado como um sistema de gerenciamento de dados unificado, que integra internamente múltiplos mecanismos de armazenamento, cada um potencialmente baseado em um modelo de dados (relacional, chave-valor, colunar, grafo etc.) [Karimov et al. 2019]. Nessa abordagem, o próprio sistema é responsável por gerenciar a distribuição e a organização dos dados entre seus componentes. Isso inclui funcionalidades como troca automática de dados entre mecanismos e particionamento baseado em carga de trabalho. Ela exige que todos os mecanismos de armazenamento estejam integrados ao núcleo do sistema, o que pode aumentar a complexidade de implementação e reduzir a flexibilidade para incorporar novos bancos externos já existentes.

Neste trabalho, são utilizados projetos que implementam uma camada de acesso. Em polystore, a camada de acesso, é frequentemente implementada como *middleware* ou *query processing layer*, é a parte arquitetural responsável pela interface de consulta unificada e pela organização de consultas sobre múltiplos modelos de armazenamento. Essa camada atua decompondo consultas, otimizando, traduzindo para as linguagens nativas dos sistemas e integrando os resultados retornados. Ela pode ser aplicada em cima de bancos de dados pré existentes, preocupando-se apenas na conexão com cada um deles.

4. Trabalhos Analisados

Os trabalhos analisados nesse artigo foram escolhidos a partir de uma busca no *Google Scholar*, utilizando a seguinte string de busca: "*multi model schema*" *AND* "*Unified language access*" *AND* "*polyglot databases*" para trabalhos publicados entre 2015 e 2025. Essa busca gerou 37 resultados, sobre estes resultados foram aplicados os seguintes critérios: (*i*) trabalhos mais relevantes de acordo com o resumo, (*ii*) qualidade do veículo de publicação e (*iii*) número de citações. Após a aplicação dos critérios de exclusão, seis trabalhos foram selecionados e são discutidos a seguir.

O primeiro trabalho trata sobre *Accio* [Wang et al. 2025], uma biblioteca *middleware* que foca em habilitar o uso de consultas poliglotas (ou federadas) em um mecanismo de consulta (*data science query engine* ou *DS Engines*). Essa camada de acesso pode ser integrada com qualquer um das *DS Engines*, melhorando a otimização e portabilidade do uso de *polystores*. Para tal, *Accio* primeiro começa a etapa de reescrita, que obtém a consulta proveniente da *DS Engine*, e analisa um plano de consulta. A segunda etapa é realizar as otimizações, incluindo o algoritmo de *pushdown* criado pelos autores. Esse algoritmo utiliza de uma abordagem iterativa, que envia para baixo os *inner joins* mais eficientes, utilizando o custo como base do cálculo:

$$benefit(R_1 \bowtie R_2) = \frac{C(fetch(R_1) \bowtie_{S_0} fetch(R_2)) - C_{in}}{C(fetch(R_1 \bowtie_{S_k} R_2)) - C_{in}}.$$

A função $benefit(R_1 \bowtie R_2)$ define o critério utilizado para decidir se um operador de junção deve ser executado por meio de um *pushdown* ou localmente após a transferência dos dados. Essa métrica expressa a razão entre o custo estimado de buscar as relações R_1 e R_2 e executar a junção em S_0, o custo de realizar a junção diretamente no componente remoto S_k e posteriormente transferir apenas o resultado. Durante esse processo, é aplicado uma interface que particiona as consultas e as transforma no formato desejado, também empregando o custo esperado.

No final da reescrita, *Accio* percorre o plano e o transforma em um plano de reescrita, formado por *querys pushdown* (que se relacionam somente com o banco de dados onde se encontram os dados necessários) e a consulta local (que junta as informações das *querys pushdown* para produzir a resposta ao cliente). Na etapa de registro, as *querys pushdown* são eviadas aos BDs correspondentes e tem seus resultados registrados na *DS Engine* utilizada. Por fim, a consulta local é executada com os resultados gerados na etapa anterior.

Diferentemente do *Accio*, *CloudMdsQL* [Kolev et al. 2016] é uma linguagem baseada em SQL que tem por objetivo acessar múltiplos BDs, sendo eles BDR ou NoSQL, com uma única consulta. Para isso, utiliza-se de subconsultas que acessam os dados na linguagem de consulta nativa de cada BD em específico. Esses, são chamados como funções e otimizados a partir de cálculos de custo. Uma subconsulta é definida como uma expressão que retorna uma tabela que contém um nome e sua assinatura (nomes e tipos das colunas). É esperado que seja definido em qual BD está o conteúdo que será retornado. Essas sub-consultas também podem ser definidas como funções em Python, possuindo parâmetros na sua assinatura.

A *query engine* para o CloudMdsQL funciona da seguinte maneira: o compilador decompõe a consulta em um plano de execução na forma de um grafo dirigido acíclico dos operadores relacionais; os nós sorvedouros correspondem as subconsultas que devem ser executadas nos BDs específicos; e a otimização ocorre por meio informações de custo, que podem ser alteradas pelo usuário quando utilizado consultas nativas. Assim como no *Accio*, a consulta é particionada e enviada somente ao BD correspondente, após terem sido otimizadas com os algoritmos de cada trabalho.

No trabalho de [Basciani et al. 2016] é apresentada a linguagem de modelagem *TyphonML* para desenvolver sistemas poliglotas híbridos. Esse trabalho difere dos demais por focar na modelagem que depois gerará a camada de acesso, e não na construção da camada em si. O projeto visa promover um ambiente para desenvolvimento de polystores. A chave principal desse projeto é o metamodelo do *TyphonML*, onde se constrói especificações para cada tipo de dado e entidade que serão conectados aos BDs. A modelagem pode ser feita utilizando tanto o editor textual quanto o gráfico. Os dois foram criados com projetos Eclipse que possuem integrações, *Xtext* e *Sirius*, respectivamente.

A segunda etapa do *TyphonML* consiste em validar a modelagem realizada pelo cliente. Utilizando de *Epsilon Object Language* (EOL) e *Epsilon Validation Language* (EVL) da plataforma *Epsilon*[2], é possível analisar e recomendar mudanças diretamente no ambiente de trabalho. Por fim, é gerado a camada de acesso para o sistema poliglota. Cada BD gera um microsserviço responsável por cuidar de todas as suas entidades. O *API Gateway* é o serviço que contém por todos os outros microsserviços e pode gerenciar relacionamentos entre diferentes tipos de BDs. Cada arquitetura possui uma especificação *OpenAI* que pode auxiliar tanto humanos quanto máquinas a utilizar o sistema.

Assim como o *Accio*, o *MSI* (*multiple sources integration* ou integração de múltiplas fontes) [Li and Gu 2019] propõem uma integração entre BDR e NoSQL em que eles possam ser acessados a partir de uma única consulta SQL. O modelo começa com o recebimento da consulta, que é enviada para o *API dispatcher*, que chama os ou-

[2]eclipse.dev/epsilon

tros métodos e envia a resposta ao cliente. O componente de *SQL Parser* é responsável por analisar sintática e lexicalmente, analisar o encapsulamento de objetos e lidar com eventuais erros. A consulta é decomposta em estruturas auxiliares, que serão utilizadas para converter a consulta em objetos da linguagem Java, e posteriormente, em consultas específicas para cada tipo de BD.

O *SQL router component* distribui e divide a consulta principal em subconsultas de acordo com o BD alvo. O componente de execução, *SQL executor*, juntamente com o adaptador, *DBMS adapter component*, enviam a subconsulta para o BD correspondente, fazendo uma verificação semântica, geração de parâmetros e checagem de erros. Após receberem o resultado, ele é enviado para o *Result merger*, que junta todos os dados obtidos utilizando o algoritmo *heap sort* e os envia de volta ao cliente. Cada um desses componentes tem acesso ao *Metadata management*, que contém as informações sobre a organização dos dados, relações, e campos de tabelas. O trabalho também propõe uma transformação semântica entre BDR e MongoDB, métodos de construção de índices do Redis e um pseudo código para que seja possível utilizar uma consulta para múltiplos BDs híbridos.

ESTOCADA [Alotaibi et al. 2019], de maneira semelhante ao *Accio* e *MSI*, é uma abordagem que permite uma aplicação utilizar dados em BDs heterogêneos, mas difere quando aceita que os dados podem estar distribuídos (uma mesma informação armazenada em mais de um banco). Além disso, utilizam o conceito de fragmentos *materialized views* para representar, por exemplo, tabelas, coleções, ou mesmo dados parciais pré-processados (como uma *cache*). Para isso, é criado a linguagem de integração QBT^{XM}, baseada na *Query Block Tree* (QBT), que suporta consultas de diferentes modelos e linguagens de dados dependendo do repositórios. Ela é baseada em um design de blocos, porém, cada bloco pode ser expressado em uma linguagem diferente e ter informações de modelos de dados diversos. Cada *materialized view* representa um esquema diferente para QBT^{XM}, que pode ter dados de vários bancos mas segue somente um tipo de armazenamento, sendo registrado em um banco correspondente.

A abordagem usa a lógica relacional conjuntiva para construir o modelo global usando as visões materializadas (fragmentos). A lógica relacional usa regras para compor as visões e as consultas QBT^{XM} são reescritas utilizando essas regras. As consultas reescritas geram um plano lógico com informações da consulta nativa de cada visão. Por fim, elas servem para descrever, de forma declarativa, como os dados podem ser obtidos e combinados, permitindo que o sistema descubra automaticamente como responder uma query.

O trabalho de [Koupil and Holubová 2022], na mesma linha de *Accio*, *MSI* e *ESTOCADA*, propõe um *framework* chamado *MM-cat* para integração e consulta de dados, mas com base na teoria das categorias. Seu objetivo é oferecer uma visão conceitual unificada, independente dos modelos de dados. O primeiro passo, assim como no *TyphonML*, consiste na criação de um esquema conceitual, na forma de um diagrama Entidade-Relacionamento (ER). Esse esquema define entidades, atributos e relacionamentos, servindo como uma visão unificada e abstrata dos dados. Em seguida, o ER é automaticamente transformado em uma *schema category*, que representa o esquema conceitual global do sistema. Nessa parte, os objetos modelam tipos conceituais de dados, como entidades, atributos e relacionamentos, enquanto os morfismos descrevem as

conexões entre esses elementos. A *schema category* garante que todas as estruturas e relacionamentos válidos estejam definidos.

Para conectar o nível conceitual ao físico, o trabalho define o *category-to-data mapping*, que especifica como objetos e morfismos da *schema category* são criados nas estruturas dos BDs. Esse mapeamento é realizado por meio de uma decomposição do esquema conceitual em múltiplos *kinds*, cada um associado a um BD e a um modelo de dados específico. Cada *mapping* define um objeto ou morfismo raiz, o BD de destino e a estrutura interna dos registros.

Após a definição dos *kinds*, o MM-cat gera automaticamente scripts contendo os comandos CREATE KIND e ALTER KIND, que podem ser executados nos BDs selecionados. Com isso, é possível adicionar dados nos bancos. Por fim, existe a *instance category*, responsável por representar os dados de fatos armazenados no sistema. Cada *instance category* segue a *schema category* e suas restrições estruturais e semânticas. O framework dessa forma consegue receber uma *query category* (que funciona como uma parte do *schema category*), definir quais informações estão em quais BDs, buscar os dados, utilizar *pullbacks* para realizar junções multi-modelo, e transformar o resultado para o modelo lógico desejado. O sistema conta com uma linguagem de consulta mais simples que as demais abordagens baseada em caminhos de acesso aos dados (datapaths). De maneira geral, é criado um grafo que representa o esquema dos dados (e onde estão armazenados) e a linguagem é baseada no caminho de acesso à informação no grafo (caminhos de morfismo).

As abordagens apresentadas demonstram técnicas diferentes para o tratamento da persistência poliglota. Embora compartilhem o objetivo de unificar o acesso a dados heterogêneos, esses trabalhos divergem em aspectos como a estratégia de otimização, o nível de abstração do esquema e o suporte à distribuição de dados. A fim de sintetizar essas diferenças e identificar as lacunas preenchidas por cada proposta, a próxima seção apresenta uma comparação entre as abordagens.

5. Discussão

Esta sessão visa discutir as diferenças e os resultados de cada um dos trabalhos analisados. Para isso, as camadas de acesso (*middlewares*) foram simplificados na Tabela 1. A tabela está organizada da seguinte forma: *Abordagem* indica a ferramenta analisada, *Ano* corresponde ao ano de lançamento, *DBs Suportados* indica quais modelos de dados são suportados, *Consulta* apresenta o tipo de linguagem de consulta utilizada e *Esquema* aponta o uso de um esquema: local, global ou nenhum.

Por serem camadas de acesso, o Accio, [Wang et al. 2025] e o MM-Cat, [Koupil and Holubová 2022], se adaptam aos modelos de BDs da *DS Engine* em que eles são aplicados. Porém, diferentemente do Accio, o MM-Cat tem como maior objetivo a criação de esquemas para depois ser criada a *query engine* a partir deles. Isso se assemelha ao TyphonML que visa criar uma linguagem de modelagem para polystores para gerar a camada de acesso após esse primeiro passo.

As principais diferenças entre esses dois projetos são: (*i*) MM-Cat cria a modelagem com base na teoria de categorias, demonstrando algoritmos para transformar os dados para as categorias e vice e versa, (*ii*) os principais componentes são os tipos conceituais e os morfismos, (*iii*) o usuário coloca cada modelo no seu BD respectivo, precisando

Tabela 1. Comparação entre Camadas de Acesso para Polystores

Abordagem	Ano	DBs Suportados	Query	Esquema
Accio	2025	DS Engine	SQL	Local
CloudMdsQL	2016	Grafo, Documento, Relacional, chave-valor	SQL-like	Nenhum
TyphonML	2016	Grafo, Documento, Relacional, chave-valor, Coluna	TyphonML	Global
MSI	2019	Documento, relational, chave-valor	SQL	Nenhum
ESTOCADA	2019	Grafo, Documento, Relacional, chave-valor	SQL-like, Redis, API, Solr PI, XQuery, XPath	Local & Global
MM-cat	2022	DS Engine	Datapaths	Global

configurar os *wrappers* (funções de conexão) de cada um e (*iv*) o armazenamento de dados só pode ser feito após a modelagem estar pronta. TyphonML utiliza um metamodelo formado principalmente por *data types*, que representam tipos de dados e *databases*, que representam os BDs ligados a esses dados.

Accio propõe uma divisão de consultas na sua otimização. Durante a fase de reescrita, tenta-se dividir a consulta de acordo com a otimização de custo. No seu algoritmo iterativo, se uma divisão for melhor que a existente, ela é substituída. Já o MSI utiliza o *SQL Parser Component* para formar uma árvore abstrata sintática. Utilizando essa árvore, é possível transformar partes da consulta em objetos de linguagem. Porém, os autores não discutem a otimização utilizando esse componente. A reescrita acontece com o algoritmo Nest-G, que visa transformar consultas complexas em consultas não aninhadas.

CloudMdsQL também utiliza da reescrita para a otimização baseada em custo. Uma consulta pode vir com subconsultas já da sua linguagem de consulta. Os autores utilizam o *bind joins* como método para realizar *semi joins* entre BDs com modelos heterogêneos, reescrevendo as subconsultas para afastar os *joins*. O ESTOCADA, por sua vez, apresenta as *materialized views* como uma das formas de otimização. A reescrita utiliza o algoritmo de *Provenance-Aware C&B algorithm (PACB)*, que tem como restrições as *materialized views* definidas. Assim como o MSI, o CloudMdsQL, ESTOCADA faz uso de um mecanismo de execução, com modelo relacional aninhado e suporta *bind joins*. A melhor reescrita é escolhida de acordo com o custo.

Alguns trabalhos se diferem de todos os restantes em certos pontos. O MSI define o uso de SQL para o *MongoBD* e métodos de construção para índices no *Redis*. ESTOCADA cria uma linguagem de integração baseada em blocos para especificar consultas entre modelos e *views*. Esse trabalho também se apoia no uso de heurísticas e teoremas como base teórica. CloudMdsQL cria sua própria linguagem de consulta. MM-Cat se baseia na teoria de categorias para modelagem. TyphonML cria as *APIs* de acesso com base no Acceleo [3].

A análise realizada indicou que a maioria dos trabalhos oferece suporte aos prin-

[3]https://eclipse.dev/acceleo/

cipais modelos de dados NoSQL, incluindo grafo, documentos, chave-valor e família de colunas, além do modelo relacional. No que se refere às linguagens de consulta, é utilizado o SQL, seja por meio do próprio padrão SQL ou de variações *SQL-like*. Sobre a estrutura, verifica-se que o uso de um esquema global é mais frequente do que esquemas locais, sendo que quase todos os trabalhos consideram algum tipo de esquema. É possível afirmar que a criação de camadas de acesso para BDs poliglotas ainda é um tópico muito estudado, e que possui diversas abordagens que podem ser utilizadas.

6. Conclusão

A revisão sistemática apresentada permitiu identificar tendências no desenvolvimento de camadas de acesso para BDs poliglotas. Observou-se que a maioria dos trabalhos oferece suporte tanto ao modelo relacional quanto aos principais modelos NoSQL (grafo, documentos, chave-valor e orientado de colunas) evidenciando a consolidação de abordagens multimodelo. Em relação às linguagens de consulta, predomina a adoção de SQL ou variações *SQL-like*, o que demonstra um esforço da comunidade em manter compatibilidade com padrões já consolidados e facilitar para usuários e desenvolvedores. Além disso, verificou-se maior uso de esquemas globais em comparação aos esquemas locais, indicando uma preferência por mecanismos que promovam visão unificada e maior nível de abstração na integração dos dados. De modo geral, os resultados apontam para a sistemas que priorizam a compatibilidade de modelos, padronização de consulta e suporte estruturado aos dados.

Apesar desses avanços, trata-se de uma área relativamente recente e em constante evolução, especialmente diante do crescimento de dados e da diversidade de modelos de armazenamento. Nesse contexto, existem amplas oportunidades de pesquisa e desenvolvimento, incluindo melhorias em técnicas de otimização distribuída, estratégias mais flexíveis de gerenciamento de esquemas e mecanismos eficientes de processamento de consultas heterogêneas. Assim, o campo apresenta significativo potencial para expansão teórica e aplicação prática nos próximos anos.

Referências

Alotaibi, R., Bursztyn, D., Deutsch, A., Manolescu, I., and Zampetakis, S. (2019). Towards scalable hybrid stores: Constraint-based rewriting to the rescue. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1660–1677. ACM.

Araujo, L., Silva, H., and Mello, R. (2025). Uma revisão sistemática simplificada e análise de estudos sobre evolução de esquemas em aplicações de persistência poliglota. In *Anais da XX ERBD*, pages 20–29, Porto Alegre, RS, Brasil. SBC.

Basciani, F., Di Rocco, J., Di Ruscio, D., Pierantonio, A., and Iovino, L. (2016). TyphonML: a modeling environment to develop hybrid polystores. In *Proceedings of the 23rd ACM/IEEE MODELS*, pages 1–5. ACM.

DB-Engines (2026). Database ranking by category. Acessado em: 12 fev. 2026.

Duggan, J., Elmore, A. J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., and Zdonik, S. (2015). The BigDAWG polystore system. *ACM SIGMOD Record*, 44(2):11–16.

Karimov, J., Rabl, T., and Markl, V. (2019). PolyBench: The first benchmark for polystores. In Nambiar, R. and Poess, M., editors, *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence*, volume 11135, pages 24–41. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

Kolev, B., Bondiombouy, C., Valduriez, P., Jimenez-Peris, R., Pau, R., and Pereira, J. (2016). The CloudMdsQL multistore system. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2113–2116. ACM.

Koupil, P. and Holubová, I. (2022). A unified representation and transformation of multimodel data using category theory. *Journal of Big Data*, 9(1):61.

Li, C. and Gu, J. (2019). An integration approach of hybrid databases based on SQL in cloud computing environment. *Software: Practice and Experience*, 49(3):401–422.

Podkorytov, M. and Gubanov, M. (2019). Hybrid.poly: A consolidated interactive analytical polystore system. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1996–1999. IEEE.

Sadalage, P. J. and Fowler, M. (2013). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Always learning. Addison-Wesley.

Sheth, A. P. and Larson, J. A. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236.

Silva, H. A. B. d., Bornia, L. G., and Mello, R. d. S. (2024). Um estudo sobre modelagem poliglota de dados. In *Escola Regional de Banco de Dados (ERBD)*, pages 11–20. SBC.

Stonebraker, M. and Cetintemel, U. (2005). "one size fits all": an idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE'05)*, pages 2–11.

Vogt, M., Stiemer, A., and Schuldt, H. (2018). Polypheny-DB: Towards a distributed and self-adaptive polystore. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3364–3373. IEEE.

Wang, X., Wang, J., Wang, T., and Zhang, Y. (2025). Accio: Bolt-on query federation. *Proceedings of the VLDB Endowment*, 18(7):2126–2135.

Yu, X., Gadepally, V., Zdonik, S., Kraska, T., and Stonebraker, M. (2019). FastDAWG: Improving data migration in the BigDAWG polystore system. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, volume 11470, pages 3–15. Springer International Publishing. Series Title: Lecture Notes in Computer Science.