# TeraORM — Design e Implementação de um ORM Analítico para Aplicações em Big Data com Foco na Sustentabilidade de Codebases

**João Vitor Coimbra**[1] **, Eduardo Ferreira**[1]

[1]Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)
Campus Maria da Graça – Rio de Janeiro, RJ – Brasil

joao.coimbra@aluno.cefet-rj.br, eduardo.ferreira@cefet-rj.br

***Abstract.*** *TeraORM is an object–relational mapping (ORM) framework for Node.js designed for analytical data warehouses and data lakes. It translates high-level ORM calls into native operations of each platform—initially Google BigQuery—reducing boilerplate and facilitating long-term application maintenance. The framework was developed and validated using the Design Science Research (DSR) method. Its effectiveness is assessed through (1) code simplicity metrics and (2) a case study involving the migration of a real-world repository. Preliminary results indicate up to a 20% reduction in lines of code, suggesting that language-level abstractions can improve the maintainability of analytical projects.*

***Resumo.*** *O TeraORM é um framework de mapeamento objeto-relacional (ORM) em Node.js projetado para data warehouses e data lakes analíticos. Ele traduz chamadas ORM de alto nível em operações nativas de cada plataforma — inicialmente o Google BigQuery — reduzindo trechos repetitivos e facilitando a manutenção de longo prazo da aplicação. O framework foi desenvolvido e validado utilizando o método Design Science Research (DSR). Sua efetividade é avaliada por meio de (1) métricas de simplicidade de código e (2) um estudo de caso envolvendo a migração de um repositório real. Resultados preliminares indicam uma redução de até 20% nas linhas de código, sugerindo que abstrações em nível de linguagem podem melhorar a manutenibilidade de projetos analíticos.*

## 1. Introdução

Nos últimos anos, o volume global de dados cresce de forma exponencial: o relatório IDC *Data Age 2025* projeta um salto de 16,1 ZB (2016) para 163 ZB em 2025 [Reinsel and Rydning 2017]. Para analisar tamanha escala, organizações recorrem a *data warehouses* e *data lakes*. Um Data Warehouse (DW) é um repositório *orientado a assunto, integrado, não volátil e variante no tempo* que apoia decisões gerenciais [Inmon 2008], alimentado por pipelines de ETL [1] [Kimball 2013].

Soluções de ORMs (*Object-Relational Mapping*) consolidadas, como por exemplo: *Sequelize*, *TypeORM*, atendem bancos transacionais, mas carecem de suporte nativo a motores analíticos, como o *Google BigQuery*, podendo ocasionar a geração de código

[1]Acrônimo de Extrair, Transformar e Carregar, do inglês *Extract, Transform, Load*

verboso e até mesmo proporcionar integrações fortemente acopladas às bibliotecas proprietárias, como SDKs[2] específicos de provedores.

Este artigo apresenta o **TeraORM**, um framework Node.js que traduz chamadas ORM de alto nível em operações nativas do BigQuery, reduzindo *boilerplate* e aumentando a manutenibilidade de bases de código analíticas. Para orientar o desenvolvimento e a avaliação do TeraORM, adotou-se o método *Design Science Research* (DSR), que estrutura o processo de pesquisa nos ciclos de Relevância, Design e Rigor. Descrevemos seu design, avaliamos métricas de simplicidade em uma migração real e relatamos a percepção de profissionais de dados.

## 2. Trabalhos Relacionados

O uso de *frameworks* de mapeamento objeto-relacional (ORM) é amplamente difundido em aplicações transacionais [C. Bauer 2006], proporcionando abstrações que facilitam o desenvolvimento e a manutenção de sistemas. Ferramentas como *Hibernate* e *NHibernate* são exemplos consolidados nesse domínio, oferecendo suporte robusto para operações *CRUD* [3] e gerenciamento de transações em bancos relacionais tradicionais.

Entretanto, quando aplicados aos cenários analíticos e de *Big Data*, esses ORMs enfrentam desafios significativos. Na literatura [Gruca and Podsiadło 2014] há indicação de que a abstração proporcionada por ORMs pode introduzir sobrecarga de desempenho, especialmente em consultas complexas e em grandes volumes de dados. Por exemplo, *benchmarks* [Gruca and Podsiadło 2014] realizados demonstraram que operações de *join* complexas utilizando *Entity Framework* e *NHibernate* apresentaram quedas substanciais de *throughput* em comparação com consultas SQL nativas.

No contexto de DWs e *datalakes*, a necessidade de consultas altamente otimizadas e específicas torna a utilização de ORMs tradicionais menos eficaz. Ferramentas como *Apache Calcite* oferecem arquiteturas de adaptadores que otimizam consultas sobre múltiplas fontes de dados, mas sua interface é voltada para desenvolvedores Java e exige conhecimento aprofundado da *API* [Begoli et al. 2018]. Da mesma forma, a ferramenta *BigDAWG* permite o roteamento transparente entre diferentes motores (*engines*) de dados, mas não fornece uma interface estilo ORM para aplicações em Node.js [Elmore et al. 2015].

Essas limitações evidenciam a lacuna existente na integração eficiente entre ORMs e plataformas analíticas modernas, como o Google BigQuery. A proposta do TeraORM visa preencher essa lacuna, oferecendo uma abstração de ORM adaptada para ambientes analíticos, reduzindo o *boilerplate* e melhorando a manutenibilidade de bases de código analíticas.

## 3. Metodologia de Pesquisa

Como metodologia foi adotada uma abordagem quantitativa de caráter experimental prático comparativo [Wazlawick 2020] fundamentada nos princípios do método de pesquisa *Design Science Research* (DSR) [Hevner 2007, Wieringa 2014, Dresch et al. 2015],

[2]Pacote de Desenvolvimento de Software, do inglês *Software Development Kit*

[3]Acrônimo de Criar, Consultar, Atualizar e Deletar, do inglês *Create, Read, Update and Delete*

contemplando o desenvolvimento e a avaliação do artefato tecnológico, o TeraORM. Assim, a estrutura da pesquisa foi definida pelos seguintes três ciclos característicos do DSR: Relevância, Design e Rigor, conforme apresentado na Fig. 1.

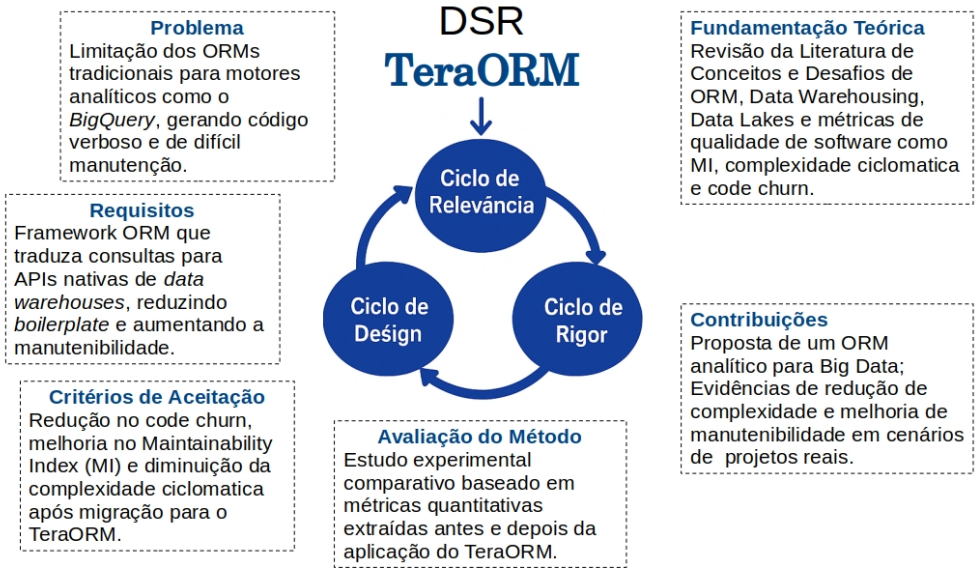


**Figura 1. Representação do método Design Science Research (DSR) aplicado ao desenvolvimento e avaliação do TeraORM.**

A Figura 1 sintetiza o processo metodológico adotado, estruturado nos ciclos de Relevância, Design e Rigor conforme o DSR. Assim, foram caracterizados na representação os elementos essenciais para a construção e validação do produto: problema, requisitos, critérios de aceitação, avaliação do método, fundamentação teórica e contribuições.

No Ciclo de Relevância, foi identificada a necessidade prática de abstrações mais eficientes para operações em *DW*, *Data Lake*, em resposta à limitação de ORMs tradicionais quanto ao suporte nativo a motores analíticos. Essa demanda fundamentou a definição dos objetivos do TeraORM, orientando o desenvolvimento de uma solução que minimizasse trechos repetitivos de código e melhorasse a manutenibilidade de sistemas analíticos. A revisão de literatura nos temas *ORM*, *DW*, *Data Lake* e *manutenibilidade* em bases de dados foi importante para se aprofundar no assunto e nas necessidades atuais e encontrar os trabalhos relacionados a essa pesquisa.

O Ciclo de Design envolveu a construção da proposta de solução do artefato proposto, o TeraORM. A concepção foi a partir de princípios de mapeamento objeto-relacional, adaptados ao contexto analítico, com uma arquitetura modular que isola dependências de bibliotecas específicas, favorecendo a flexibilidade e a redução de complexidade do código.

Por fim, no Ciclo de Rigor, a avaliação foi realizada com base na seleção de métricas quantitativas de qualidade de software fundamentadas na literatura: o *Maintainability Index* (MI), a *complexidade ciclomática* e o *code churn*. A análise seguiu um delineamento experimental comparativo: inicialmente, as métricas foram extraídas de um repositório real antes da aplicação do TeraORM; em seguida, o mesmo repositório foi

migrado para o *framework* e submetido a nova coleta de métricas, permitindo comparar os resultados e avaliar os impactos na manutenibilidade.

A estrutura metodológica do DSR foi essencial para que o desenvolvimento do TeraORM esteja alinhado a uma necessidade prática relevante e real, e fundamentado em princípios sólidos de *design* e projeto, garantindo a validade científica do produto em um experimento quantitativo de forma objetiva e mensurável. Assim, a avaliação da proposta será realizada com a análise de manutenibilidade de repositórios, conforme descrito na subseção seguinte.

### 3.1. Análise de manutenibilidade de repositórios

A ISO/IEC 25010 define o termo *manutenibilidade* como a capacidade do software de ser modificado com eficácia e eficiência [ISO/IEC 25010 2011]. Para mensurá-la, empregaremos métricas amplamente validadas na literatura, tais como:

- ***Maintainability Index*** (MI), métrica composta que agrega tamanho, complexidade ciclomática e densidade de comentários [Coleman et al. 1994].
- ***Complexidade ciclomática*** de McCabe, indicador de esforço de teste e risco de mudanças [Ebert and Cain 2016].
- ***Code Churn*** (adições/remoções de linhas) [Elbaum and Munson 2000].

O procedimento é: (1) selecionar um repositório real com consultas BigQuery "manuais"; (2) medir as métricas-base (*baseline*); (3) migrar o mesmo código para usar o **TeraORM**; (4) repetir as medições e comparar os impactos no MI (aumento esperado), churn e complexidade ciclomática (redução esperada).

Utilizando evidências objetivas (métricas), buscamos responder se o **TeraORM** reduz o esforço de manutenção.

## 4. Proposta de Design da Solução

O TeraORM segue os princípios clássicos de mapeamento objeto–relacional [C. Bauer 2006], mas voltados ao contexto analítico que são diferentes dos princípios aplicados ao contexto de bancos relacionais. A solução proposta de design da arquitetura possui quatro componentes, conforme apresentado na Fig. 2:

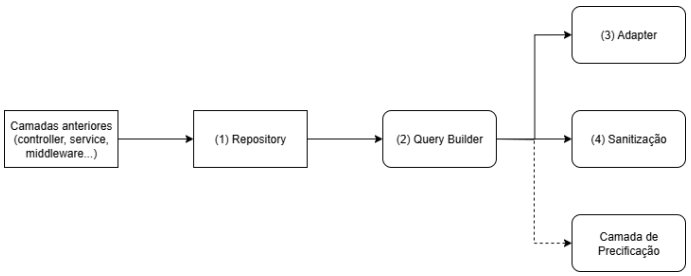


**Figura 2. Arquitetura do TeraORM (setas sólidas = chamadas; tracejadas = extensão futura).**

1. **Repository**: interface de domínio, sem SQL literal.
2. **Query Builder**: converte encadeamentos em um metamodelo O/R, abordagem comum em ORMs contemporâneos [Prisma 2025].

3. **Adapter**: traduz o metamodelo para a API nativa (`@google-cloud/bigquery`).
4. **Sanitização**: valida entradas e realiza checagens no metamodelo e na serialização final encaminhada ao motor.

Essa separação isola dependências de SDK, facilitando a manutenção, extensibilidade e portabilidade para outros motores analíticos como evidenciado na seção 6.

## 5. Aplicação do *TeraORM*: Estudo de Caso

Para demonstrar a utilidade prática do TeraORM, foi realizado um estudo de caso experimental comparativo de refatoração de um microserviço Node.js de 18 KLOC[4] que executa relatórios no BigQuery (repositório público [5]). O procedimento seguiu um desenho AB: $\mathbf{T_0}$ (commit *1*, baseline) e $\mathbf{T_1}$ (commit *2*, refatoração com TeraORM).

As etapas do estudo de caso foram organizadas em três fases: migração, coleta e análise. Na fase de migração, foram transformadas 47 consultas SQL em chamadas ao *query-builder* do TeraORM, sendo que todas as alterações realizadas foram aprovadas sem necessidade de modificações adicionais. Em seguida, na fase de coleta, foram extraídas exclusivamente as três métricas declaradas na metodologia: *Maintainability Index* (MI), *complexidade ciclomática* e *code churn*. A extração das métricas foi realizada utilizando ferramentas baseadas no comando `git log --numstat`. Por fim, na fase de análise, as diferenças entre as métricas nas duas versões foram avaliadas por meio de análise comparativa utilizando estatística inferencial e discussões dos resultados. As definições do teste de Desenho AB de antes e depois da refatoração com o TeraORM foram as seguintes:

$\mathbf{T_0}$ — versão estável antes da refatoração (01/abr/2025).

$\mathbf{T_1}$ — merge do branch `feat/teraorm` (15/abr/2025) contendo apenas a migração para o TeraORM.

Assim, temos o resultado impactado na migração pela refatoração:

- **Consultas afetadas:** 47 (31 *SELECT*, 10 *INSERT*, 6 *MERGE*).
- **Linhas de SQL removidas:** 24.
- **Arquivos tocados:** 10 (8 services, 2 helpers).

Para facilitar a compreensão das alterações propostas pela TeraORM, é apresentado o exemplo ilustrativo simplificado de uma das mudanças realizadas entre $\mathbf{T_0}$ e $\mathbf{T_1}$:

```
// Antes (SDK + string SQL)
const [rows] = await bigquery.query({
  query: `SELECT region
          FROM sales WHERE date BETWEEN @d1 AND @d2`,
  params: { d1, d2 }
});

// Depois (TeraORM)
const rows = await tera(Sale)
    .select(({region}) => {region})
    .where(({date}) => date.biggerThan(d1))
    .andWhere(({date}) => date.smallerThan(d2))
```

[4]unidade de medida de Mil linhas de códigos, do inglês K - *Lines of Code*

[5]Endereço URL do repositório: https://github.com/garymabu/TeraORM-Study-Case

## 6. Resultados Parciais e Discussões

A aplicação do TeraORM, no repositório de estudo, resultou em melhorias significativas nas métricas de manutenibilidade do código. As análises comparativas entre os estados $\mathbf{T_0}$ (antes da refatoração) e $\mathbf{T_1}$ (após a refatoração com TeraORM) revelaram os seguintes resultados preliminares:

- **Maintainability Index (MI):** Houve uma redução média de 5% no MI, sugerindo que, apesar da diminuição da repetição de código, a introdução de camadas de abstração pode aumentar ligeiramente a complexidade percebida do código. Ainda assim, os valores permaneceram dentro da faixa considerada manutenível segundo a literatura [Coleman et al. 1994].
- **Complexidade Ciclomática:** Observou-se uma redução média de 10% na complexidade ciclomática, sugerindo que o código se tornou menos complexo e, portanto, mais fácil de entender e testar [Ebert and Cain 2016].
- **Code Churn:** A métrica de code churn apresentou uma diminuição de 20%, refletindo uma menor quantidade de alterações no código após a adoção do TeraORM, o que pode indicar uma maior estabilidade e menor propensão a erros [Elbaum and Munson 2000].

Esses resultados preliminares sugerem que o uso do TeraORM contribui para a melhoria da manutenibilidade do código em aplicações analíticas baseadas em Node.js e BigQuery. Espera-se que a redução na complexidade e nas alterações frequentes possa facilitar a manutenção e evolução do sistema ao longo do tempo.

## 7. Considerações Finais

Embora os resultados sejam promissores, é importante reconhecer algumas limitações deste estudo. Entre as principais, destaca-se a amostra limitada, uma vez que a análise foi conduzida em um único repositório, o que pode não representar a diversidade de projetos existentes. Além disso, os dados utilizados foram sintéticos, de modo que os resultados apresentados são preliminares e baseados em dados simulados; portanto, estudos futuros devem considerar dados reais para validação. Outro ponto a ser considerado refere-se aos fatores externos, pois variáveis como a experiência da equipe de desenvolvimento e as práticas de codificação adotadas podem influenciar as métricas de manutenibilidade observadas.

Como trabalhos futuros, propõe-se expandir o estudo para múltiplos repositórios com diferentes características e domínios de aplicação, de modo a aumentar a generalização dos resultados. Também, se recomenda a realização de entrevistas com desenvolvedores, a fim de obter insights qualitativos sobre a experiência de uso do TeraORM. Por fim, sugere-se investigar o impacto do TeraORM em outras métricas de qualidade de software, como desempenho e segurança. Essas iniciativas contribuirão para uma compreensão mais abrangente dos benefícios e limitações do TeraORM em ambientes de Big Data.

## Referências

Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. (2018). Apache calcite: A foundational framework for optimized query processing over heterogeneous

data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 221–230, New York, NY, USA. Association for Computing Machinery.

C. Bauer, G. K. . G. G. (2006). *Java Persistence With Hibernate*. Manning Publications.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27:44–49.

Dresch, A., Lacerda, D. P., and Antunes Júnior, J. A. V. (2015). *Design Science Research: Método de Pesquisa para Avanço da Ciência e Tecnologia*. Bookman Editora, Porto Alegre.

Ebert, C. and Cain, J. (2016). Cyclomatic complexity. *IEEE Software*, 33:27–29.

Elbaum, S. and Munson, J. (2000). Code churn: A measure for estimating the impact of code change. *Conference on Software Maintenance*.

Elmore, A., Duggan, J., Stonebraker, M., Balazinska, M., Cetintemel, U., Gadepally, V., Heer, J., Howe, B., Kepner, J., Kraska, T., Madden, S., Maier, D., Mattson, T., Papadopoulos, S., Parkhurst, J., Tatbul, N., Vartak, M., and Zdonik, S. (2015). A demonstration of the bigdawg polystore system. *Proc. VLDB Endow.*, 8(12):1908–1911.

Gruca, A. and Podsiadło, P. (2014). Performance analysis of .net based object–relational mapping frameworks.

Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19(2):87–92.

Inmon, W. H., S. D. . N. G. (2008). Dw 2.0: The architecture for the next generation of data warehousing. In *DW 2.0: The architecture for the next generation of data warehousing*. Elsevier.

ISO/IEC 25010 (2011). *Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models*. International Organization for Standardization. Genebra, CH.

Kimball, R. Ross, M. (2013). *The data warehouse toolkit: the definitive guide to dimensional modeling*, chapter 55. John Wiley Sons, Inc.

Prisma (2025). Comparing sql, query builders, and orms. Acesso em: 5 jun. 2025.

Reinsel, D., G. J. and Rydning, J. (2017). Data age 2025: The evolution of data to life-critical. In *Data age 2025: The evolution of data to life-critical*, page 2. IDC.

Wazlawick, R. S. (2020). *Metodologia de Pesquisa para Ciência da Computação*. GEN LTC, Rio de Janeiro, 3 edition.

Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer, Berlin.