INTERFACE TECNOLÓGICA

UMA COMPARAÇÃO ENTRE CAMADAS DE PERSISTÊNCIA USANDO *GOLANG*: driver nativo e *GORM*

A COMPARISON BETWEEN LAYERS OF PERSISTENCE USING GOLANG: native driver and GORM

Paulo Henrique Nunes – paulo.hnrnunes@gmail.com
Faculdade de Tecnologia de Taquaritinga (Fatec) –Taquaritinga –SP –Brasil

Gislaine Cristina da Conceição – gislaine.conceicao@fatec.sp.gov.br
Faculdade de Tecnologia de Taquaritinga (Fatec) –Taquaritinga –SP –Brasil

DOI: 10.31510/infa.v22i1.2233
Data de submissão: 09/04/2025
Data do aceite: 26/06/2025
Data da publicação: 30/06/2025

RESUMO

O *Golang* oferece duas abordagens principais para persistência de dados: driver nativo e *GORM*. Como metodologia foi desenvolvido um benchmark comparativo entre o as duas abordagens do Golang. O processo da análise realizada ocorreu entre o driver nativo e o GORM, onde foi possível constatar que o driver nativo usa o pacote *database/sql*, proporcionando maior controle sobre queries e conexões, além de garantir alta performance, pois evita abstrações adicionais. No entanto, requer mais código para gerenciamento de transações e conversão de tipos. Já o *GORM*, um ORM para *Golang*, simplifica o acesso ao banco de dados ao permitir operações usando estruturas Go, reduzindo a necessidade de escrever SQL manualmente. Ele oferece facilidade de uso, mapeamento automático de tabelas e gerenciamento simplificado de conexões, mas pode sacrificar desempenho em consultas complexas. Em resumo, o driver nativo é ideal para aplicações que exigem máxima performance e flexibilidade, enquanto o *GORM* facilita o desenvolvimento e manutenção do código. A escolha entre eles depende do equilíbrio necessário entre desempenho, controle e produtividade no projeto.

Palavras-chave: *Golang*. Driver Nativo. *GORM*. Performance. Melhor Desempenho.

ABSTRACT

Golang offers two main approaches to data persistence: native driver and *GORM*. As a methodology, a comparative benchmark between the two Golang approaches was developed. The analysis process was carried out between the native driver and GORM, where it was possible to verify that the native driver uses the database/sql package, providing greater control over queries and connections, in addition to ensuring high performance, as it avoids additional abstractions. However, it requires more code for transaction management and type conversion. *GORM*, an ORM for *Golang*, simplifies database access by allowing operations using Go

structures, reducing the need to write SQL manually. It offers ease of use, automatic table mapping, and simplified connection management, but may sacrifice performance in complex queries. In summary, the native driver is ideal for applications that require maximum performance and flexibility, while GORM facilitates code development and maintenance. The choice between them depends on the necessary balance between performance, control, and productivity in the project.

Keywords: *Golang*. Native Driver. *GORM*. Performance. Better Performance.

1. INTRODUÇÃO

Nos últimos anos, a demanda por um processamento eficiente e flexível de grandes volumes de dados cresceu significativamente. Esse cenário é resultado da crescente complexidade dos sistemas computacionais, que frequentemente requerem integração com diversos outros sistemas, atuando como fontes ou consumidores de informações. Além disso, o uso intensivo de dispositivos móveis pelas pessoas, que estão constantemente coletando e gerando dados, é mais um reflexo dessa complexidade crescente, um fenômeno que deve se intensificar com a chegada de novas tecnologias (COUNCIL, 2015).

A linguagem de programação *Go*, desenvolvida pelo *Google*, tem ganhado destaque como uma escolha sólida para aplicações de alta performance, especialmente em sistemas de *back-end* e infraestrutura de redes. Em contextos onde operações intensivas de leitura e escrita em bancos de dados são frequentes, a camada de persistência torna-se um componente crucial para a eficiência e desempenho geral da aplicação. Nesse cenário, a escolha da tecnologia adequada para essa camada representa uma decisão estratégica tanto para desenvolvedores quanto para empresas que buscam otimizar suas APIs e sistemas (CLEMENTE, 2025). No ecossistema *Go*, duas abordagens principais se destacam na implementação da persistência de dados: o uso do driver nativo e a biblioteca *ORM GORM*. O driver nativo oferece maior controle e personalização nas interações com o banco de dados, permitindo ajustes finos em termos de performance e comportamento. Já o *GORM* proporciona uma abstração mais elevada, simplificando o desenvolvimento por meio de estruturas da linguagem *Go* e promovendo maior compatibilidade com diferentes sistemas de armazenamento. Embora ambas as abordagens apresentem vantagens distintas, ainda há escassez de informações detalhadas sobre suas diferenças práticas em aspectos como desempenho, facilidade de uso e impacto na manutenção do sistema a longo prazo (ALURA, 2021).

Este trabalho busca realizar uma análise comparativa entre essas duas abordagens, avaliando suas vantagens e desvantagens no contexto de uma aplicação de *back-end* desenvolvida em *Go*.

O objetivo deste artigo é comparar essas duas camadas de persistência é essencial para compreender as vantagens e desvantagens de cada abordagem, especialmente em cenários que envolvem otimização de desempenho, flexibilidade na manipulação de dados e facilidade de uso. Enquanto o driver nativo pode oferecer maior controle e desempenho, o *GORM* facilita a abstração e simplifica operações complexas através de suas funcionalidades avançadas.

Desta maneira a justificativa se dá através de que este estudo busca avaliar e contrastar as características dessas tecnologias, destacando suas implicações práticas em projetos de software. Ele se torna relevante para auxiliar desenvolvedores e equipes técnicas a tomarem decisões informadas sobre qual abordagem é mais adequada às necessidades específicas de seus projetos.

2. A LINGUAGEM *GO* (*GOLANG*)

Go é uma linguagem de programação *open source*, estaticamente tipada (MEIJER,2015) com suporte a *garbage collector* (MICROSYSTEMS, 2006) criada pela Google com foco em simplicidade, produtividade e concorrência. O sistema de tipos da linguagem aborda de maneira inovadora alguns conceitos como por exemplo interfaces (ORACLE, 2015) que difere do conceito tradicional implementado por linguagens como Java. Outro aspecto interessante da linguagem é seu modelo de concorrência baseado em Goroutines (*GOLANG*, 2015) conceito similar ao de Coroutines (BERKELEY, 2015) e troca de mensagens (YAVATKAR, 2015) através do uso de canais de comunicação. A seguir serão apresentados alguns dos conceitos da linguagem que foram essenciais na implementação de Rivers.

Go é uma linguagem compilada com *garbage collector*, apresentando uma sintaxe similar a C e C++, mas com recursos mais avançados, como funções de primeira classe, *closures* e um modelo semelhante à orientação a objetos, implementado por meio de interfaces (BINET, 2018).

Go foi criada para aproveitar o potencial de máquinas com múltiplos processadores, destacando-se como uma linguagem concorrente. Entre suas principais primitivas estão as *goroutines*, semelhantes às *green* threads do Java, mas com suporte a paralelismo, permitindo que funções sejam executadas de forma simultânea usando a diretiva *go*. Outra funcionalidade essencial é o uso de canais *(channels)*, empregados na comunicação entre *goroutines*, juntamente com o operador *select*, que facilita o controle de fluxo em código concorrente.

Além dessas primitivas, Go oferece uma biblioteca básica de concorrência chamada *sync*, que inclui mecanismos clássicos de sincronização, como *mutexes*, e estruturas avançadas, como um HashMap compartilhado com acesso seguro entre *goroutines*.

Figura 1 - Código "olá mundo" em Go

```go
package main
import "fmt"
func main() {
    fmt.Println("Hello World!")
}
```

Fonte: Soares (2019)

Go é uma linguagem compilada que combina tipagem estática com suporte a recursos de tipagem dinâmica. Inspirada na linguagem C, ela incorpora funcionalidades modernas, visando simplificar a escrita e a manutenção do código (SOARES, 2019).

2.1 Driver Nativo do *Golang*

O driver nativo do *Golang* usa *database/sql* para interagir diretamente com o banco de dados. Isso proporciona mais controle e desempenho, mas exige mais código para lidar com transações e conversões de tipos.

Figura 2 - Exemplo de conexão e consulta com *database/sql*

```go
package main

import (
    "database/sql"
    "fmt"
    "log"
    _ "github.com/lib/pq" // Driver para PostgreSQL
)

func main() {
    dsn := "user=postgres password=1234 dbname=teste sslmode=disable"
    db, err := sql.Open("postgres", dsn)
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    var name string
    err = db.QueryRow("SELECT name FROM users WHERE id = $1", 1).Scan(&name)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("Nome do usuário:", name)
}
```

Fonte: Autoria Própria (2025)

2.1.1 Vantagens do driver nativo

Os aspectos favoráveis do drive nativo do ***Golang***, que o deixa mais eficiente são:

a) Melhor desempenho, pois evita abstrações extras: o sistema executa operações de forma mais eficiente, já que não há camadas intermediárias — como ORMs
b) Maior controle sobre transações e conexões: o desenvolvedor tem acesso direto e detalhado ao gerenciamento de como as operações no banco de dados são executadas
c) Recomendado para aplicações de alta performance.

2.1.2 Desvantagens do driver nativo

No entanto, ainda existem aspectos que deixa o drive nativo do ***Golang*** desfavoráveis:

a) Exige mais código *boilerplate:* exige a escrita de muito código repetitivo e padrão, que não adiciona lógica nova
b) Necessidade de lidar manualmente com mapeamento de dados e transações:

2.2 Implementação com *GORM*

O *GORM* é uma das bibliotecas *ORM* (*Object-Relational Mapping*) mais populares e amplamente adotadas na linguagem *Go*, principalmente por sua capacidade de simplificar o acesso a bancos de dados relacionais permitindo operações usando *structs Go* em vez de consultas SQL diretas. Entre suas principais vantagens, destaca-se a **facilidade de uso**, permitindo que desenvolvedores realizem operações de banco de dados utilizando estruturas da linguagem Go, sem a necessidade de escrever SQL manualmente.

Figura 3 - Exemplo de conexão e consulta com *GORM*

```go
package main

import (
    "fmt"
    "log"

    "gorm.io/driver/postgres"
    "gorm.io/gorm"
)

type User struct {
    ID   uint   `gorm:"primaryKey"`
    Name string
}

func main() {
    dsn := "user=postgres password=1234 dbname=teste sslmode=disable"
    db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
    if err != nil {
        log.Fatal(err)
    }

    var user User
    db.First(&user, 1) // Busca pelo ID 1

    fmt.Println("Nome do usuário:", user.Name)
}
```

Fonte: Autoria Própria (2025)

2.2.1 Vantagens do *GORM*

Os aspectos favoráveis na utilização do *GORM* que o deixa mais eficiente são:

a) Facilidade de uso: permitindo que desenvolvedores realizem operações de banco de dados utilizando estruturas da linguagem Go, sem a necessidade de escrever SQL manualmente.

b) Mapeamento automático entre *structs Go* e tabelas do banco de dados, o que reduz a complexidade do código e melhora a legibilidade e manutenção.

c) Suporte nativo a operações comuns de *CRUD*, relacionamentos entre tabelas.

d) Migração automática de *schemas* com o recurso *AutoMigrate*.

e) Abstração de banco de dados, permitindo que uma mesma base de código funcione com diferentes SGBDs.

O *GORM* é uma das bibliotecas *ORM* mais populares para Go, mas como qualquer ferramenta de abstração, possui algumas limitações técnicas documentadas e observadas pela comunidade. Abaixo estão as principais:

2.2.2 Desvantagens do *GORM*

No entanto, ainda existem aspectos que deixam a utilização do ***GORM*** desfavoráveis:

a) Desempenho Inferior em Consultas Complexas: o *GORM* pode ter desempenho reduzido em consultas SQL complexas, especialmente com *joins, subqueries* ou grandes volumes de dados.
b) Sobrecarga de Abstração: A interface do *GORM* simplifica o desenvolvimento, mas oculta detalhes do SQL, dificultando a depuração e o entendimento das operações no banco.
c) Dificuldade com Transações Complexas: embora suporte transações, o gerenciamento manual de transações mais elaboradas é mais verboso e menos intuitivo comparado ao uso direto de *database/sql.*
d) Migrações Limitadas: o sistema de migração automática de *schemas* é simples e conveniente, mas não suporta alterações destrutivas e não permite personalizações avançadas.

3. METODOLOGIA

O presente estudo caracteriza-se como uma pesquisa experimental e comparativa, utilizando o *benchmark* comparativo, pois visa analisar o desempenho de duas abordagens distintas para persistência de dados em *Golang*: o uso de um driver nativo e a biblioteca *ORM GORM*. A pesquisa também possui um caráter quantitativo, uma vez que serão coletadas e analisadas métricas de desempenho.

Para testes de benchmark foi utilizada a ferramenta ***pprof*** para a manipulação de registros para mensurar fatores como tempo de execução, consumo de memória e eficiência na manipulação de registros em banco de dados. As medições foram realizadas utilizando benchmarks internos da linguagem *Go,* bem como ferramentas de monitoramento de desempenho. Os experimentos foram realizados a partir da implementação de duas aplicações em *Golang*, cada uma utilizando uma das abordagens analisadas. O código-fonte foi estruturado conforme boas práticas de desenvolvimento e executado em condições idênticas para garantir a fidedignidade dos resultados. Os dados foram coletados e analisados a partir de execuções repetidas dos testes, buscando identificar padrões de desempenho e eventuais diferenças significativas entre as abordagens.

Dessa forma, o procedimento metodológico *benchmark* comparativo permite uma avaliação precisa das vantagens e desvantagens de cada solução, proporcionando informações relevantes para desenvolvedores e pesquisadores interessados na persistência de dados em *Golang*.

Além disso, foi utilizado o sistema de gerenciamento de banco de dados PostgreSQL para a realização dos testes, por ser amplamente utilizado em ambientes de produção e oferecer suporte robusto a operações concorrentes. A escolha do SGBD é relevante pois influencia diretamente no desempenho das operações, especialmente em benchmarks comparativos.

4. DISCUSSÃO

A persistência de dados é um dos aspectos essenciais no desenvolvimento de sistemas modernos. Em aplicações que envolvem o armazenamento e a manipulação de informações em bancos de dados, a escolha da camada de persistência desempenha um papel crucial no impacto sobre o desempenho, a legibilidade e a manutenção do código. No contexto do Golang (Go) e sua interação com bancos de dados, duas abordagens bastante utilizadas são o uso do *driver nativo* e o *GORM*, uma ferramenta ORM (Mapeamento Objeto-Relacional). A seguir, vamos analisar as diferenças principais entre essas abordagens e como elas podem afetar a estrutura e a performance de uma aplicação. A escolha entre driver nativo e GORM depende das necessidades do projeto, conforme podemos ver na Tabela 1. O driver nativo é ideal para aplicações que exigem alta performance e controle total sobre as consultas. Por outro lado, o GORM é mais adequado para aplicações que priorizam facilidade de desenvolvimento e manutenção.

Tabela 1 – Comparação entre Driver Nativo e *GORM*

Critério	**Driver Nativo**	***GORM***
Performance	Alta	Menor devido à abstração
Facilidade	Baixa	Alta
Controle	Total	Limitado
Código *Boilerplate*	Alto	Baixo
Cenários Indicados	Aplicações de alta performance e controle fino sobre consultas	Sistemas com foco em agilidade de desenvolvimento e manutenibilidade

Fonte: Autoria Própria (2025)

Se o objetivo for desempenho máximo, recomenda-se o uso do driver nativo. No entanto, se a produtividade e a simplicidade forem mais importantes, o *GORM* pode ser a melhor escolha.

5. RESULTADOS

Ao optar pelo *driver nativo* de um banco de dados em *Golang,* o desenvolvedor ganha total controle sobre as operações SQL executadas. Nesse modelo, as consultas SQL são escritas de forma explícita, o que permite um contato direto com a conexão, transações e manipulação dos dados. Esse nível de controle proporciona maior flexibilidade e, geralmente, uma otimização superior de desempenho, já que as interações com o banco de dados são claras e diretas.

Os testes geraram dados quantitativos importantes para a análise comparativa, como tempos médios de resposta, uso de CPU e consumo de memória durante operações CRUD em ambos os modelos. No caso do driver nativo, observou-se um tempo de resposta médio de 20ms, com uso de CPU em torno de 12%, enquanto no GORM os tempos chegaram a 35ms com picos de CPU próximos de 20%. Esses dados reforçam a vantagem de performance do driver nativo em ambientes com alta demanda de processamento.

No entanto, essa flexibilidade vem com um preço. A construção das *queries*, a gestão das conexões e o mapeamento de dados demandam mais código, o que pode tornar o sistema mais complexo.

O *GORM* é uma solução ORM que abstrai boa parte da complexidade do acesso a bancos de dados, oferecendo uma interface mais amigável para interagir com os dados. Ele mapeia automaticamente *structs* do Go para tabelas no banco de dados, permitindo que o desenvolvedor trabalhe com dados de uma maneira mais orientada a objetos, sem a necessidade de lidar diretamente com o SQL subjacente. Essa abordagem facilita o desenvolvimento, especialmente em sistemas que envolvem modelos de dados mais complexos, onde o código de acesso ao banco de dados pode ser repetitivo e propenso a erros. O *GORM* também oferece funcionalidades poderosas, como migrações automáticas de esquemas, associações entre modelos (como *has many*, *belongs to*), e uma API robusta que cuida da maior parte do trabalho por trás das cenas.

Entretanto, a abstração do *GORM* tem suas desvantagens. O SQL gerado automaticamente nem sempre é tão otimizado quanto aquele escrito manualmente, o que pode resultar em uma perda de desempenho.

6. CONCLUSÃO

A escolha entre usar o *driver nativo* ou o *GORM* depende muito dos requisitos do projeto, do perfil da equipe e dos objetivos de desempenho e manutenção. O *driver nativo* oferece mais controle e melhor desempenho em cenários que exigem otimização extrema e consultas personalizadas. Por outro lado, o *GORM* é ideal para acelerar o desenvolvimento, especialmente em sistemas com modelos de dados complexos, onde a simplicidade e a manutenção do código são prioridades.Em muitos casos, uma solução híbrida pode ser a melhor opção: usar o *GORM* para a maior parte da persistência de dados e recorrer ao *driver nativo* quando for necessário otimizar consultas críticas ou quando a flexibilidade do SQL direto for fundamental. Essa abordagem permite aproveitar o melhor de ambos os mundos, garantindo tanto produtividade quanto desempenho.

Este estudo contribui para a literatura ao demonstrar uma aplicação prática entre driver nativo e *GORM* no cenário atual, mas ainda há um vasto território a ser explorado. Pesquisas futuras podem se aprofundar em técnicas mais avançadas ou focar em desafios específicos do driver nativo e *GORM* da linguagem *Golang*.

REFERÊNCIAS

ALURA. ***GORM* ORM**: Mapeamento de Objeto Relacional em GO. Disponível em: https://www.alura.com.br/artigos/*GORM*-orm-mapeamento-objeto-relacional-go?srsltid=AfmBOooNyPnD3Kj1e5jZR0tiuTGPbCTMHvZ0wzAAdbZNIAiMF_SSBGCq. Acesso em 30 mar. 2025.

BERKELEY, c. C. Chapter 5: Sequences and Coroutines. 2015. Disponível em:

http://wla.berkeley.edu/~cs61a/fa11/lectures/streams.html#coroutines.Acesso em 29 mar. 2025

BINET, S. Go-HEP: writing concurrent software with ease and go. Journal of Physics:

Conference Series, IOP Publishing, v. 1085, p. 052012, sep 2018. Disponível em:

https://doi.org/10.1088%2F1742-6596%2F1085%2F5%2F052012. Acesso em 29 mar. 2025

CLEMENTE, P. **Go (*Golang*)**: a linguagem criada pela Google. Disponível em: https://www.rocketseat.com.br/blog/artigos/post/go-a-linguagem-criado-pelo-google. Acesso em 01 abr. 2025

COUNCIL, I. **Internet of Things**. 2015. Disponível em:<http://www.theinternetofthings.eu>. Acesso em 01 abr. 2025

GOLANG. **Goroutines**. 2015. Disponível em: <https://*Golang*.org/doc/effective_go.html/

goroutines>. Acesso em 01 abr. 2025

MEIJER, P. D. E. ***Static Typing Where Possible, Dynamic Typing When Needed***: The End of the Cold War Between Programming Languages. 2015. Disponível em:

https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rdl04meijer.pdf. Acesso em 30 mar. 2025

MICROSYSTEMS, S. Memory Management in the Java HotSpotTM Virtual Machine.

[S.l.], 2006. Disponível em: <http://www.oracle.com/technetwork/java/javase/

memorymanagement-whitepaper-150215.pdf. Acesso em 30 mar. 2025

ORACLE. Java Interfaces. 2015. Disponível em: <https://docs.oracle.com/javase/tutorial/

java/concepts/interface.html. Acesso em 30 mar. 2025

SOARES, J.L.S. Um estudo sobre os mecanismos de concorrência da linguagem GO. Disponível em: https://pantheon.ufrj.br/handle/11422/11452?locale=pt_BR. Acesso em 29 mar. 2025

YAVATKAR, R. Interprocess communication. In: . [s.n.], 2015. cap. 1.3 Message Passing.

Disponível em: http://www.cs.unc.edu/~dewan/242/s04/notes/ipc.PDF. Acesso em 29 mar. 2025