Go言語におけるGORMとRawSQLのデータベース操作性能比較

ICT活用総合実習 最終発表

中西 悠元 ／ 20122055

研究概要

研究テーマ（目的）

Go言語におけるORMライブラリ（GORM）とRaw SQLの性能を定量的に比較し、実務での技術選択指針を提供する

背景

ORMは開発効率を向上させるが、性能オーバーヘッドが指摘されている。しかし、Go言語における体系的な比較研究は少ない。

アプローチ

実際のWebアプリケーションを想定した包括的なベンチマーク実験により、CRUD操作・複雑なクエリ・メモリ効率を多角的に分析

ORMとは？

定義

ORM（Object-Relational Mapping）= オブジェクト関係マッピング

プログラムのオブジェクト（構造体）とデータベースのテーブルを自動的に対応付ける技術。
SQLを直接書かずに、プログラムの言語でデータベースを操作できるようにするライブラリ。

🔷 GORM（ORM）：Go言語の構造体を使ってデータベースを操作
例: db.Where("age >= ?", 20).Find(&users)

🔶 Raw SQL: SQL文を直接記述してデータベースを操作
例: SELECT * FROM users WHERE age >= 20

実装例：GORMとRaw SQLの比較

［例］年齢が20歳以上のユーザーをデータベースから取得する処理

🔷 GORM（ORM）

```go
func main() {
    type User struct {
    ID   uint
    Name string
    Age  int
}

// GORMで取得
var users []User
db.Where("age >= ?", 20).Find(&users)

}
```

✅ 特徴

- コードが簡潔（2行で完結）
- SQLを書く必要がない
- 型安全で読みやすい

🔶 Raw SQL

```go
func main() {
    type User struct {
    ID   uint
    Name string
    Age  int
}

// Raw SQLで取得
var users []User
rows, _ := db.Raw("SELECT * FROM users WHERE age >= ?", 20).Rows()
defer rows.Close()
for rows.Next() {
    var u User
    db.ScanRows(rows, &u)
    users = append(users, u)
}
```

✅ 特徴

- SQLを直接制御できる
- より詳細な制御が可能
- コード量が多い（7行）

GORMのメリット

開発効率の向上

SQLを書く必要がなく、Go言語の構造体だけでDB操作が可能。コード量が削減され、開発スピードが向上。

型安全性

Go言語の型システムを活用し、コンパイル時にエラーを検出。SQLインジェクション対策も自動化。

自動マイグレーション

構造体の定義からテーブルを自動生成・更新。データベーススキーマの管理が容易に。

リレーション管理

テーブル間の関連（1対多、多対多）を構造体で表現。関連データの取得が直感的に記述可能。

フック機能

データ保存前後に自動処理を実行。バリデーション、ログ記録、キャッシュ更新などが容易。

測定している主な評価指標

①実行時間

- 平均実行時間（Avg）→ 全実行の平均値
- 中央値（Median）→ 外れ値の影響を受けにくい
- 最小値/最大値（Min/Max）→ 最速・最遅ケース
- P95/P99 → 上位5%・1%の遅延

② 実行時間のばらつき（安定性）

- 標準偏差（StdDev）
- 変動係数（CV = StdDev / Avg）

処理時間の安定性 → CVが小さいほど安定

③ スループット

- 平均スループット（ops/sec）

1秒間に何件処理できるか → 高負荷時の処理能力

④ メモリ使用量

- 総メモリ使用量（MB）

ORMの内部処理・GC負荷 → サーバーコスト影響

⑤ 成功率・信頼性

- 成功率（%）、総操作数、総実行時間

エラーや失敗が発生していないか確認

実験環境

💻 ハードウェア環境

- 機種: MacBook Pro (2020, 13-inch)
- 機種ID: MacBookPro17,1
- プロセッサ: Apple M1チップ
  8コア（パフォーマンス×4、効率×4）
- メモリ: 16 GB ユニファイドメモリ
- ストレージ: 内蔵SSD

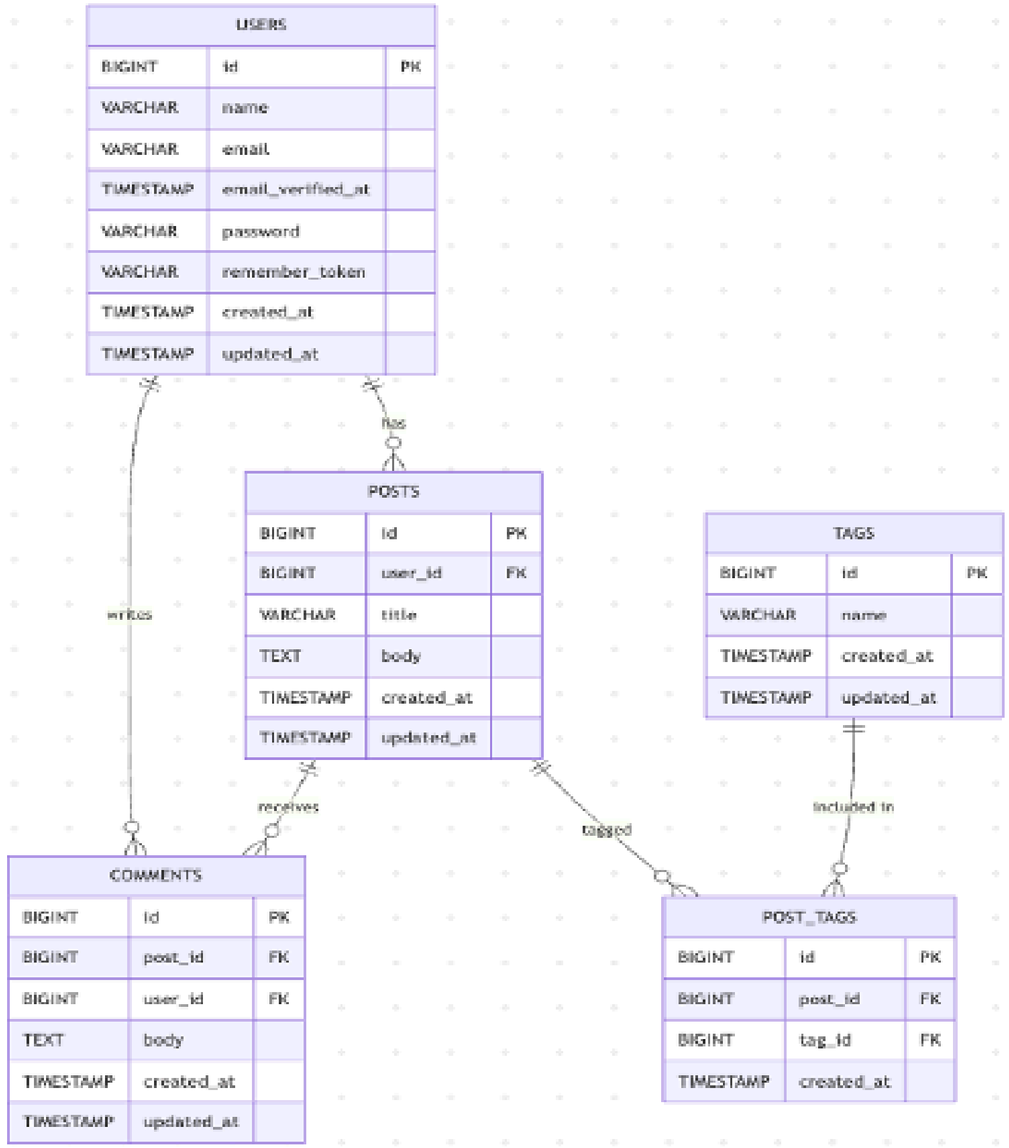
⚙️ ソフトウェア環境

- プログラミング言語: Go 1.24.4
- ORMライブラリ: GORM v1.30.2
- データベースドライバ: go-sql-driver/mysql v1.8.1
- データベース: MySQL 8.0 (Dockerコンテナ)
- テストデータ生成: gofakeit v6.28.0

実験条件統一のための工夫

データベースをDockerコンテナで実行することで、実験環境を再現可能にし、ネットワークやストレージの影響を最小化。テストデータはgofakeitで自動生成し、データの一貫性を確保。

実験環境



users

ブログの利用者。記事の投稿やコメントを行う。

posts

ブログ記事。タイトル・本文・公開状態を持つ。

comments

記事へのコメント。返信によるネスト構造に対応。

tags

記事を分類するラベル。複数の記事に付与可能。

post_tags

記事とタグの多対多を実現する中間テーブル。

ベンチマーク測定項目

CRUD操作（8種類）

- User: Create/Read/Update/Delete
- Post: Create/Read/Update/Delete

複雑なクエリ（5種類）

Pagination, N+1最適化, JOIN, 集計（GROUP BY）, IN句

測定設定: 反復回数10回、データ量1,000～10,000レコード

測定した5つの複雑なクエリパターン

Pagination（ページ分割表示）

大量データを少しずつ表示する処理
LIMIT・OFFSETを使った効率的なデータ取得

```
SELECT * FROM posts
 LIMIT 10 OFFSET 20
```

N+1最適化（関連データの効率的取得）

関連テーブルのデータを効率的に取得
1回の親クエリ + N回の子クエリを最適化

```
# 悪い例: 1+N回のクエリ
# 良い例: PreloadやJOINで
一括取得
```

JOIN（複数テーブルの結合）

複数のテーブルを結合してデータを取得
外部キーで関連付けられたデータの取得

```
SELECT users.*, posts.*
 FROM users JOIN posts
 ONusers.id=posts.user_id
```

集計（グループごとの計算）

GROUP BYやCOUNT、SUMなどを使った集計処理
カテゴリ別の統計情報を取得

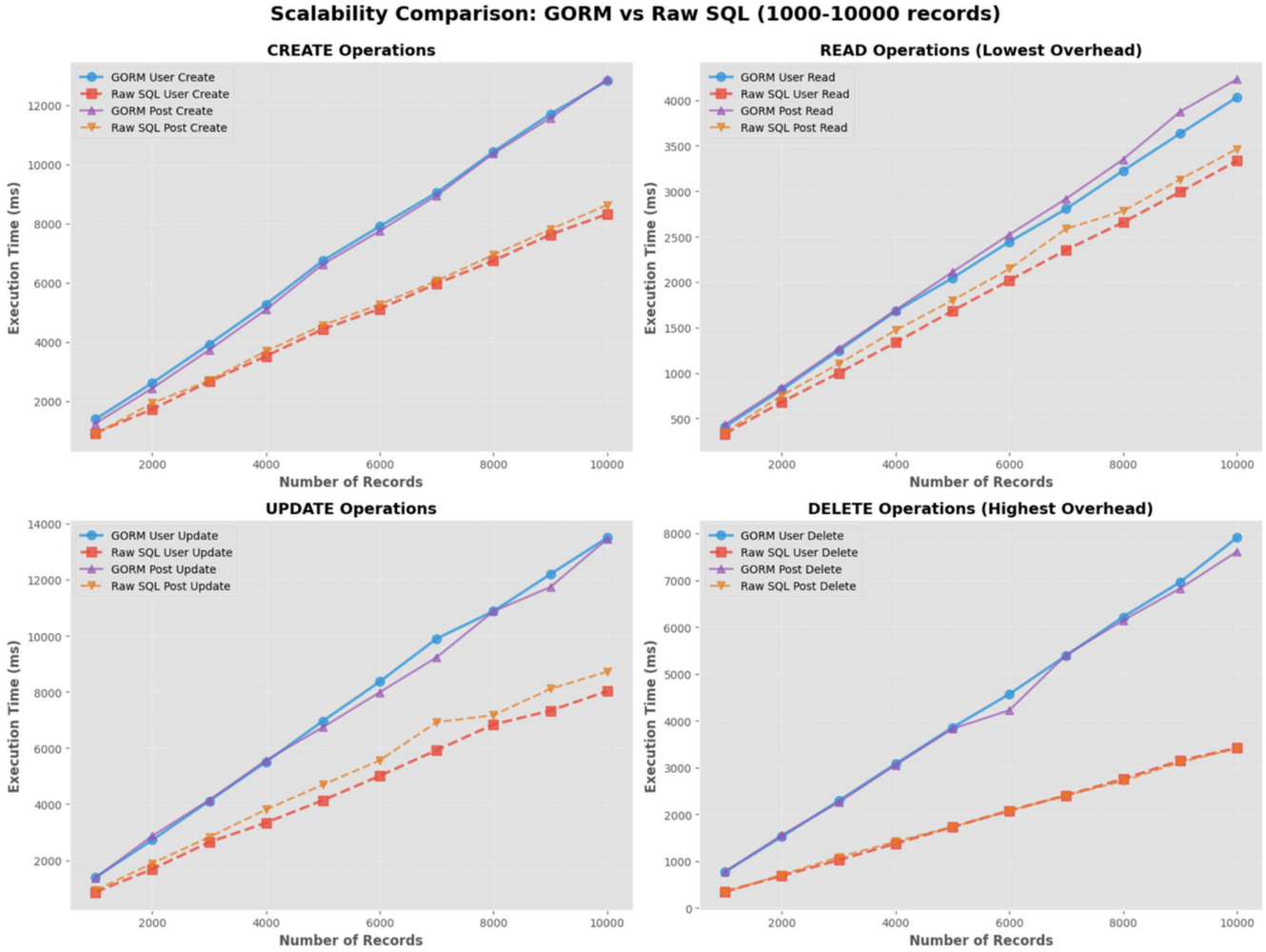
```
SELECT user_id, COUNT(*)
 FROM posts
 GROUP BY user_id
```

IN句（大量ID指定検索）

複数のIDを指定してデータを取得
大量の条件を一度に検索する処理

```
SELECT * FROM users
 WHERE id IN (1, 2, 3, ..., 100
```

結果1：スケーラビリティ比較（1,000〜10,000レコード）



主要な発見

✅ **CREATE操作**

Raw SQLが一貫して40-50%高速。データ量増加に対して両者ともに線形に増加。

✅ **READ操作**

最も差が小さい領域。Raw SQLが約20%高速だが、実用上は許容範囲。

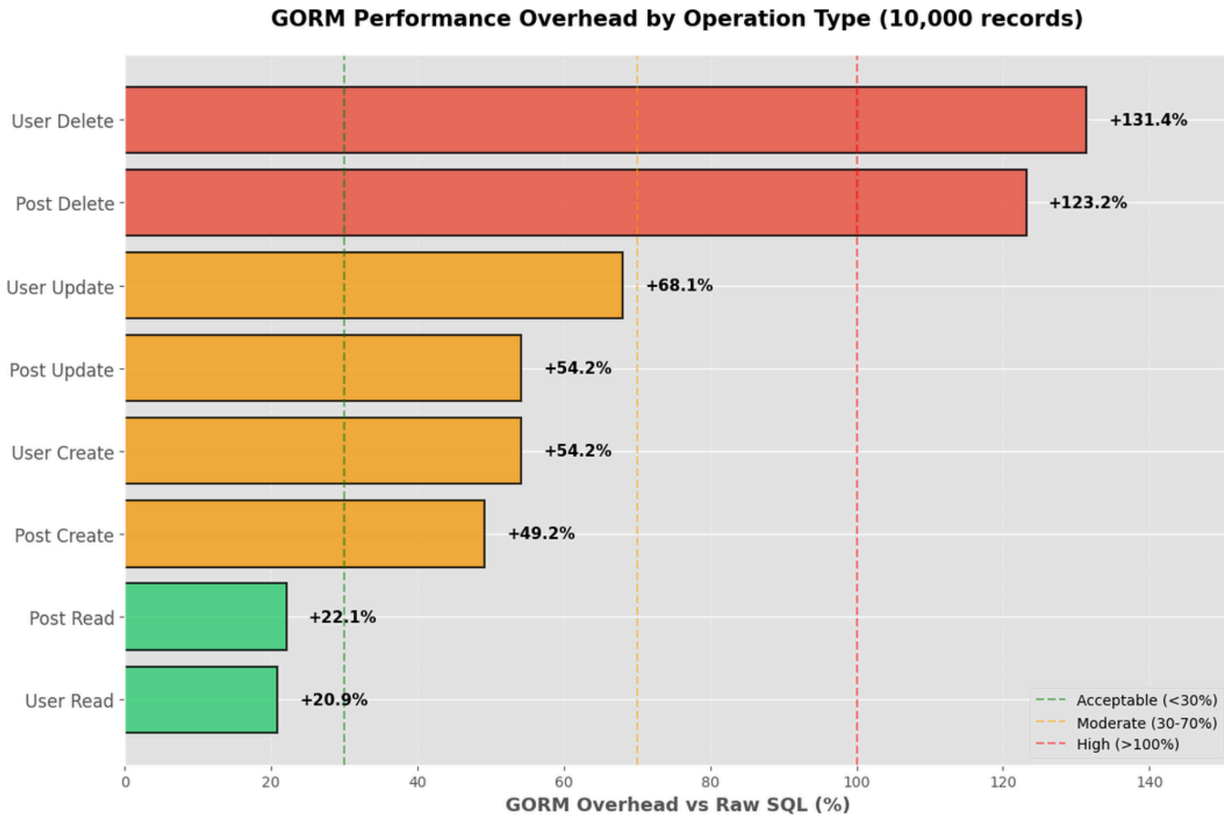
✅ **UPDATE操作**

Raw SQLが50-60%高速。GORMのバリデーション処理がオーバーヘッド。

⚠️ **DELETE操作**

最も差が大きい。Raw SQLが2倍以上高速。GORMのソフトデリート処理が影響。

結果2：GORMのオーバーヘッド率（10,000レコード）



主要な発見

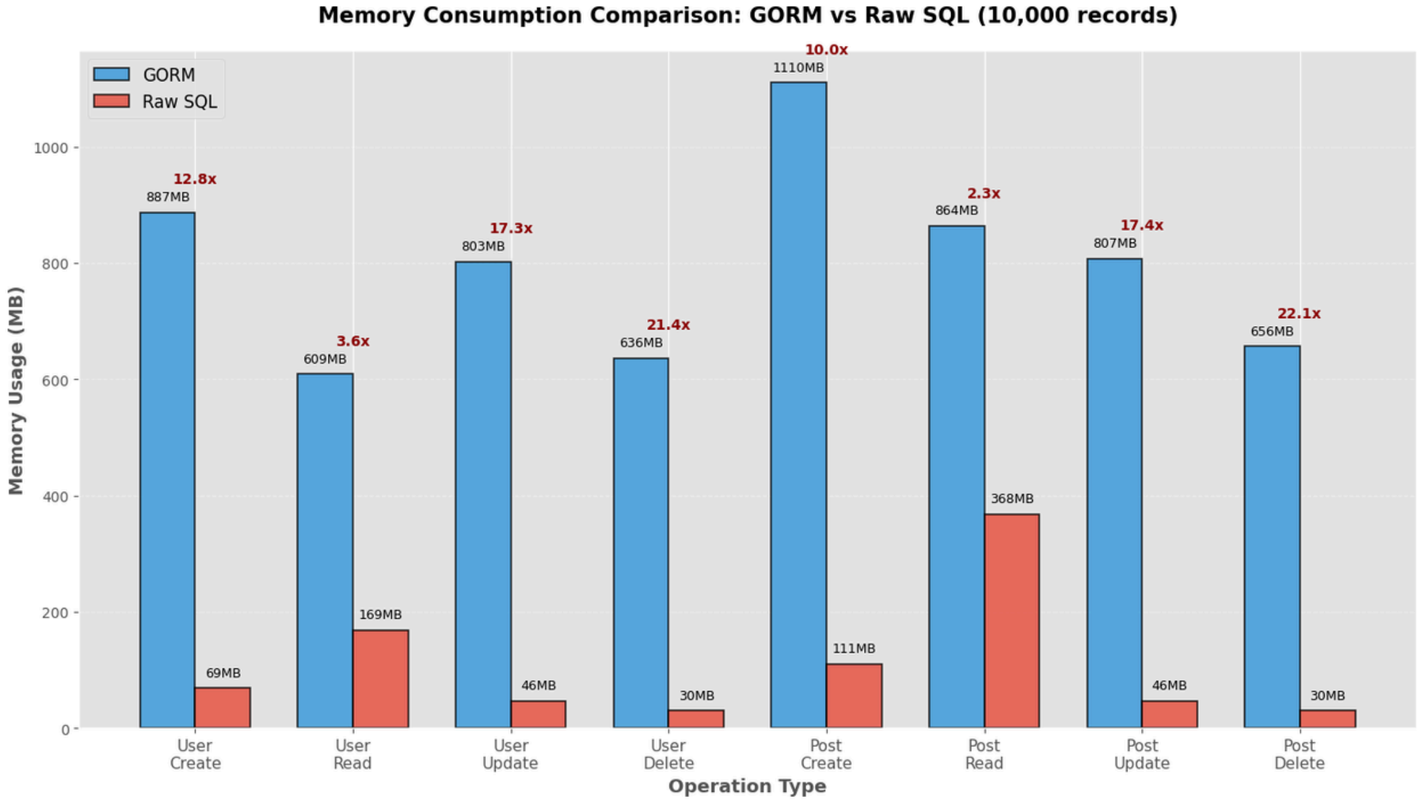
✅ **許容範囲（<30%）**
- User Read: +20.9%
- Post Read: +22.1%

⚠️ **中程度（30-70%）**
- Create/Update: +49〜68%

❌ **高い（>100%）**
- User Delete: +131.4%
- Post Delete: +123.2%

結論: 読み取りはGORMでも実用的。書き込み・削除は要検討。

結果3：メモリ使用量比較（10,000レコード）



主要な発見

メモリ効率の差

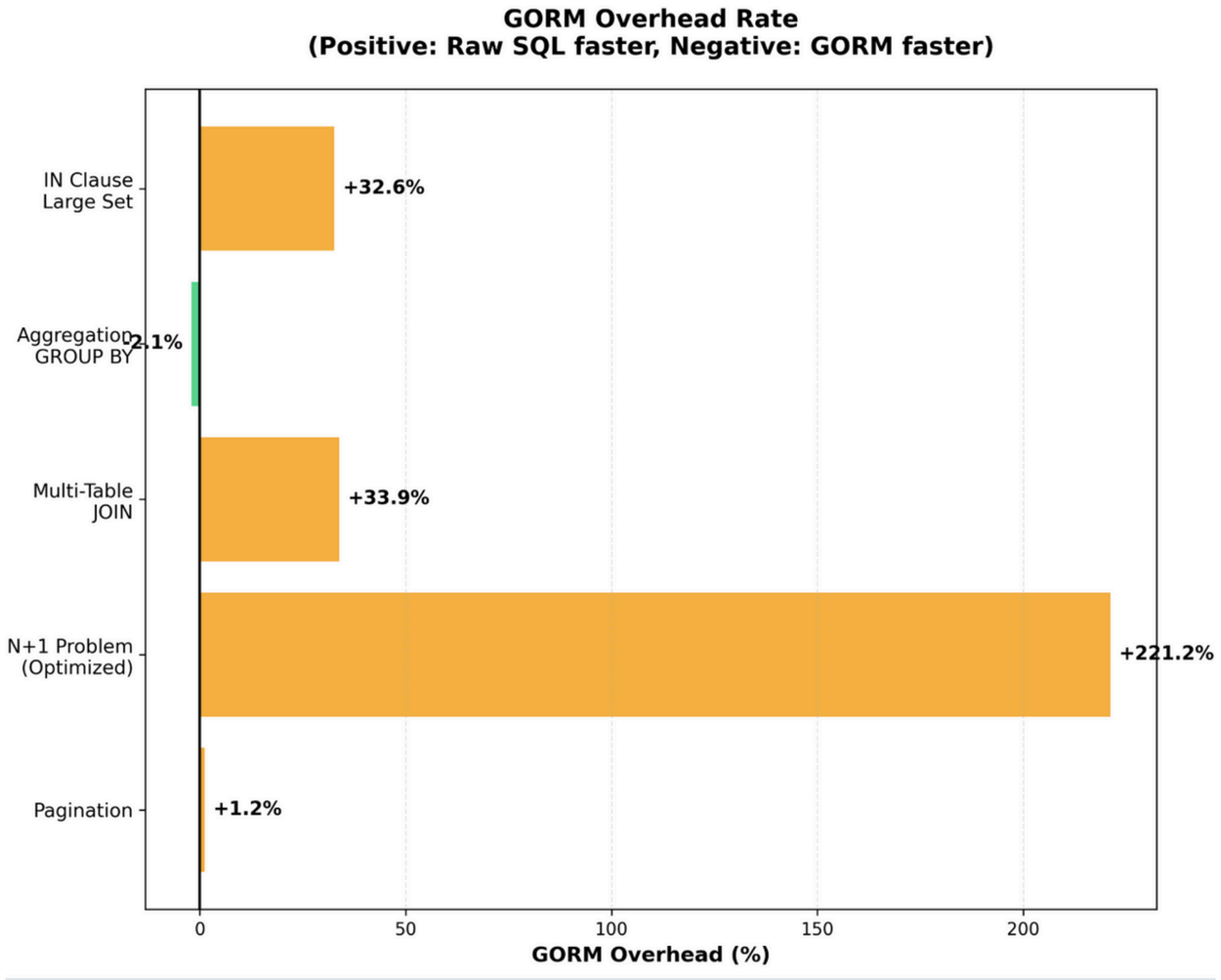
- User Create: 12.8倍
- User Update: 17.3倍
- User Delete: 21.4倍

⚠️ GORMは全操作でRaw SQLの3〜20倍以上のメモリを消費

💡 重要

大規模データ処理やメモリ制約環境ではRaw SQLが必須。

結果4：複雑なクエリのオーバーヘッド率



主要な発見

✅ Pagination: +1.2%
ほぼ同等の性能

⚠️ IN Clause: +32.6%
中程度のオーバーヘッド

⚠️ Multi-Table JOIN: +33.9%
Raw SQLの制御性が優位

❌ N+1 Problem: +221.2%
最適化しても3倍以上遅い

考察1：なぜGORMは遅くなるのか？

①リフレクションによるオーバーヘッド

GORMは実行時にリフレクションを用いて構造体の型情報を解析し、SQLを動的に生成する。
この過程で、フィールド走査・型変換・バインド処理が発生し、Raw SQLより処理コストが高くなる。

② フック・バリデーション処理の自動実行

GORMはCRUD操作時にフック関数、バリデーション、関連レコード処理を自動的に実行する。
不要な処理も常に実行されるため、操作ごとのオーバーヘッドが増加する。

③ メモリアロケーションの増加

GORMは型情報キャッシュやクエリビルダーなどの中間オブジェクトを内部で多く生成する。
その結果、Raw SQLより多くのメモリを消費する。

考察2：操作タイプ別の性能差の解釈

N+1問題で3倍の差がつく理由

GORMのPreloadは2段階クエリ（親→子）を実行するが、Raw SQLの最適化された
JOINには及ばない。クエリ回数は削減できても、GORMの中間処理（結果のマッピング、
関連付け）が追加コスト。 → ブログの記事一覧（記事+著者）など、関連データ表示で顕著。

DELETE操作で差が最大になる理由

削除操作でGORMが特に遅い理由：

- ソフトデリート：実際は削除せず、deleted_atフラグを更新（余分なUPDATE）
- 関連レコードの自動処理：外部キー制約のチェックと関連削除
- BeforeDelete/AfterDeleteフックの実行、トランザクション管理

→ 定期的な古いデータの削除、テストデータのクリーンアップなど、大量削除が必要な場面ではRaw SQLで一括削除する方が圧倒的に効率的。

-結論 -

Raw SQLを選択すべきケース

①高スループットが求められるシステム

- APIサーバー、リアルタイム処理
- 書き込み・削除が頻繁な用途

②パフォーマンスクリティカルな処理

- バッチ処理
- データ移行・ETL処理

③メモリ効率が重要な環境

- コンテナ化された環境
- リソース制約のあるシステム

④複雑なクエリを多用する場合

- 複数テーブルのJOIN
- N+1問題が発生しやすいデータ構造

-結論 -

GORMを選択すべきケース

①開発速度・保守性を重視

- スタートアップ、MVPフェーズ
- 機能追加が頻繁なプロジェクト

②複雑なビジネスロジック

- フック、バリデーション、ソフトデリートなどを活用
- 関連データの自動管理が必要

③チーム開発

- SQLスキルにばらつきがあるチーム
- コードの統一性・可読性を重視

④パフォーマンス要件が緩い場合

- 読み取り中心のアプリケーション
- 小〜中規模のデータ量

ご清聴ありがとうございました

補足資料1：リフレクション（Reflection）とは？

定義

リフレクション = プログラムが実行時に自分自身の構造を調べ、操作する機能

構造体 User に ID,Name、Age というフィールドがあれば、GORMは実行時にリフレクションを使ってこれらのフィールドを検出し、対応するSQL（SELECT id,name, age FROM users）を自動生成する。

```
type User struct {
ID   uint
Name string
Age  int
}
```

→

SELECT id,name, age FROM users

補足資料2：フック処理（Hook）とは？

定義

フック = データベース操作の前後に自動実行される処理

GORMでは、データの作成・更新・削除などの操作の前後に、自動的に実行される処理を定義できる。

主なフックの種類：
- BeforeCreate: データ作成前に実行（例: IDの自動生成、タイムスタンプ設定）
- AfterCreate: データ作成後に実行（例: 作成ログの記録、通知送信）
- BeforeUpdate: データ更新前に実行（例: 変更履歴の保存、バリデーション）
- BeforeDelete: データ削除前に実行（例: 関連データのチェック、削除ログ）

補足資料3：バリデーション（Validation）とは？

定義

バリデーション＝データの妥当性をチェックする処理

データベースに保存する前に、データが正しい形式・値であることを確認する処理。
GORMでは、構造体のタグを使ってバリデーションルールを定義でき、保存時に自動的にチェックされる。

主なバリデーション例：
- 必須チェック: 必ず値が入力されているか確認（例: ユーザー名が空でないか）
- 文字数制限: 文字数が範囲内か確認（例: パスワードが8文字以上）
- 形式チェック: 正しい形式か確認（例: メールアドレスの形式）
- 範囲チェック: 値が範囲内か確認（例: 年齢が0〜150の範囲）