Building an ORM for Go

Eyal Posener

Go Israel Meetup 04/01/2018

I’m Eyal

- I Work at Stratoscale (we’re hiring)
- I love:
  - Go
  - Open source
  - Programming
  - Creating stuff
  - Surfing

Agenda

- Using SQL with Go
- What is an ORM?
- Existing ORMs
- Why Existing ORMs are not Good Enough?
- What Kind of ORM do I Propose?
- How do I Propose to do It?

Using SQL with Go

Demo:

`sql/sql.go`

What is an ORM?

- Interacting with relational databases is very common for us developers.
- Usually we do the same old sh#@$...
- ORM comes to make our life easier!
- Basically:
  - Database access with programming language functions.
  - Fills the [Program object] < == > [SQL table] gap.

Something like:

```
wife.Save()
wife.Get(1) // get by id
```

Existing ORMs

Actually there are so many...

The most common is GORM

Demo:

`gorm/gorm.go`

Why GORM is not Good Enough?

This is great! - Short and working API to interact with our DB!

But...

- Type safety! wild values.

```
func (s *DB) Related(value interface{}, foreignKeys ...string) *DB
```

  - Is `db.Related(&MyModel, “alice”, “bob”)` is valid, what will it return?
  - Find myself checking examples and documentation all the time...
  - Find out in run-time
- Lacks concurrency support

```
$ go test ./... -race
...
FAIL
```

Why GORM is not Good Enough?

- API is unclear
  - Always looking for examples and documentations
  - I'm not always sure if there was a DB transaction
- Slow (uses reflections)
- A bit weird error handling:

```
err := db.Delete(&person).Error
```

  - Not go-ish
  - Trust the developer to remember to check the error.
  - Easily miss an error.
  - Unable to mock

What do I propose?

- Type safety in Go? code generation!
- Run a command that generates an ORM code for a type:

```
$ orm -type Person
```

- The generated code has ORM functionality, but it is
  - Typed
  - Simple
  - Expressive

How do I Propose to do it?

Basic demo (from repository examples):

https://github.com/posener/orm/examples/simple/

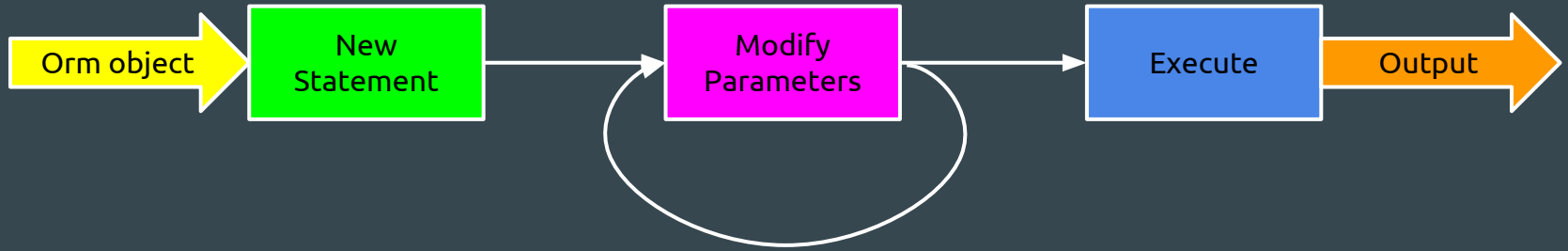
Code Generation

1. Load the code, using the Package Loader package.
2. Lookup:
    a. Find the type
    b. Read it's fields
    c. Find their type.
3. Create a relations graph (one-to-one, one-to-many).
4. Generate code accordingly.

Guidelines:

- Less generated code / more runtime.
- Human readable generated code
- Isolate dialect differences.

Generated Code / Statement flow




```
Items, err := o.Select().GroupBy(...).Where(...).Query()
```

Generated Code / Builder Pattern

Builder Pattern

- We have SQL statements: CREATE, SELECT, INSERT...
- Each statement has parameters:
  - SELECT: table, columns, where, group by, order by, limit, joins, ...
  - INSERT: table, columns and values, ...
- Make parameters private and expose "parameter builders".

```
o.Select().GroupBy(ColName)
```

```
func (o *ORM) Select() *SelectBuilder {
      return &SelectBuilder{
            ...
      }
}

type SelectBuilder struct {
      params SelectParams
}

func (b *SelectBuilder) GroupBy(col Col) *SelectBuilder {
      b.params.GroupBy.Add(col)
      return b
}

type SelectParams struct {
    Table   string
    Columns []string
    ...
}
```

The Generated Code

- Exec/Query function:
  a. Builds an SQL command.
  b. Send it to the SQL driver.
  c. Process the returned values.

Relations: Go Vs. SQL

	Go	SQL

One to One (Forward relation)	`type A struct {`
	`  B *B`
	`}`
	
	`type B struct {`
	`  ID int64`
	`}`
		`CREATE TABLE b (`
		`  id INT PRIMARY KEY`
		`);`
		`CREATE TABLE a (`
		`  b_id INT,`
		`  FOREIGN KEY (b_id) REFERENCES b(id)`
		`);`
		
		`SELECT * FROM a LEFT JOIN b ON a.b_id=b.id;`

One to Many (Reversed relation)	`type A struct {`
	`  ID int64`
	`  B  []B`
	`}`
	
	`Type B struct {`
	`  A *A  // or AID int64`
	`}`
		`CREATE TABLE a (`
		`  id INT PRIMARY KEY`
		`);`
		`CREATE TABLE b (`
		`  a_id INT,`
		`  FOREIGN KEY (a_id) REFERENCES a(id)`
		`);`
		
		`SELECT * FROM a LEFT JOIN b ON a.id=b.a_id;`

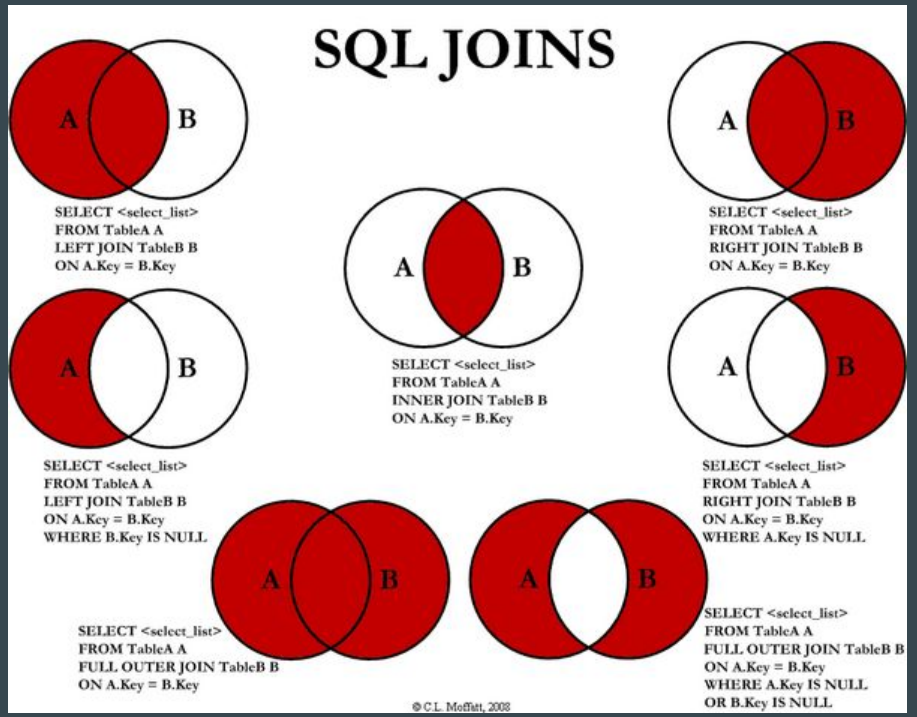
Relations: Go Vs. SQL

	Go	SQL

Many to Many	`type A struct {`
	`  ID int64`
	`  B  []B`
	`}`
	
	`Type B struct {`
	`  ID int64`
	`  A  []A`
	`}`
		`CREATE TABLE a (`
		`  id INT PRIMARY KEY`
		`);`
		`CREATE TABLE b (`
		`  id INT PRIMARY KEY`
		`);`
		`CREATE TABLE ab_relation (`
		`  a_id INT,`
		`  b_id INT,`
		`  FOREIGN KEY (a_id) REFERENCES a(id),`
		`  FOREIGN KEY (b_id) REFERENCES b(id)`
		`);`
		
		`SELECT * from a`
		`  LEFT JOIN ab_relation ON a.id=ab_relation.a_id`
		`  LEFT JOIN b ON ab_relation.b_id=b.id;`

Demo?

`relations/*.sql`



Relation in ORM: Information from 2 Types??

- Need to query according to the joined type
- Need to parse the joined columns also.

```
as, err := leftORM.Select().JoinRight(rightORM.Select().Joiner()).Query()
```

- The **joiner** is a Go interface! :-D
  - Used to build the SQL statement,
  - And also to parse the returned SQL rows.

Relations: Left's ORM

SelectBuilder can Join Right if it gets an object that implements the required interface

```
type rightJoiner interface {
    Params() runtime.SelectParams
    Scan(dialect string, values []driver.Value) (*Right, int, error)
}

func (sb *SelectBuilder) JoinRight(joiner rightJoiner) *SelectBuilder
```

Relations: Right's ORM

SelectBuilder exposes a Joiner function which returns an object that implements the required interface.

```
type Joiner interface {
    Params() runtime.SelectParams
    Scan(dialect string, values []driver.Value) (*Right, int, error)
}

type joiner struct {
    builder *SelectBuilder
}

func (j *joiner) Params() runtime.SelectParams {
    return j.builder.params
}

func (j *joiner) Scan(dialect string, values []driver.Value) (*Right, int, error)
{
    return j.builder.scan(dialect, values)
}

func (b *SelectBuilder) Joiner() Joiner {
    return &Joiner{builder: b}
}
```

Demo

Examples from orm package:

https://github.com/posener/orm/tree/master/examples

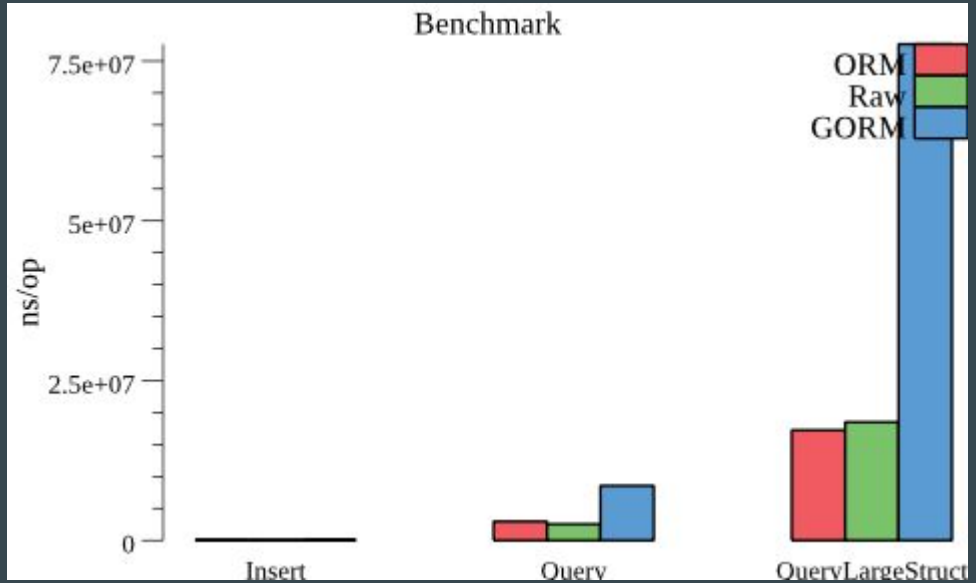
Migrations

- AutoMigration: look at the current table and transform it to the required state.
  - Risky
  - Can implement only simple operations (to reduce risk)
    - Add nullable column
    - Add foreign key
- Migrations:
  - Forward and backward (called up/down)
  - Versioned
  - Can do complicated operations
  - Save state in SQL itself
  - SQL script that we run manually.

Performance

ORM's performance is similar to the raw SQL function.

GORM's performance are very poor.



Challenges

- A lot of work!
- Different dialects.
- Template for generated code is not fun.
- A lot of other solutions.

Good Resources

http://go-database-sql.org/

Thank You!

Eyal Posener

posener@gmail.com

https://github.com/posener

https://github.com/posener/orm