Nikolay Kuznetsov

Software Engineering @ Zalando Finland

sqlc deep dive: how SQL becomes type-safe Go code



About me

- Software Engineering @ Zalando Helsinki 🇫🇮
- Author of *pgx-outbox* and *sqlc++* projects
- Contributed to ***sqlc***
- Also into *Postgres* and *Kubernetes* communities
- 3rd time at GoLab, 2nd time as a speaker



Fun fact

Your session sqlc deep dive: how SQL becomes type-safe Go code has been accepted

Inbox

G GoLab 2026 (via S... 10.29

to me

Hi Nikolay Kuznetsov,

According to our Editorial Team evaluations, your talk session sqlc deep dive: how SQL becomes type-safe Go code has been accepted for GoLab 2025: congrats!

We just need your confirmation to add you to the conference schedule: if you want to confirm your presence to our conference, please click the button below.

Confirm participation

Once you've confirmed your presence, our colleague Ambra will contact you to define your accommodation.

Further details can be found at the Welcome Speakers page.



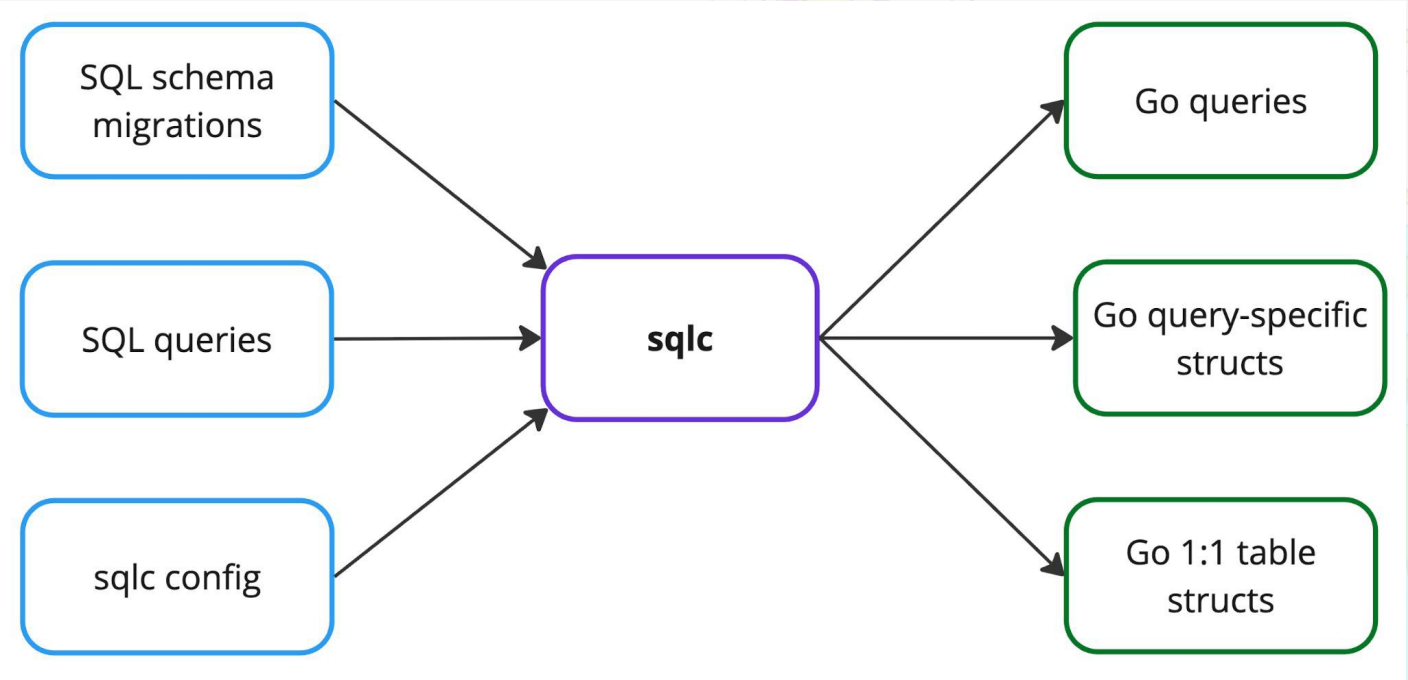
- There is no limit how deep is sqlc



Intro to sqlc

- Deterministic code generator
- SQL-first approach
- Schema-aware
- Compile-time type safety
- Written in Go

sqlc in a nutshell



Talk focus on

- **Postgres** (MySQL, MariaDB, ClickHouse, SQLite)
- **Go** generated code (JSON, other languages using plugins)
- ***pgx*** driver and API (lib/pq)
- **DB-mode** is preferred over in-memory mode

Schema migration

```
CREATE TABLE IF NOT EXISTS cart_items
(
    owner_id   BIGINT                   NOT NULL REFERENCES users (id),
    product_id UUID                     NOT NULL,
    price      NUMERIC(10, 2)           NOT NULL,
    currency   VARCHAR(3)               NOT NULL,
    quantity   INT         DEFAULT 1    NOT NULL CHECK (quantity > 0),
    created_at TIMESTAMPTZ DEFAULT now() NOT NULL,
    PRIMARY KEY (owner_id, product_id)
);
```

Query to generate

```
-- name: AddItem :one
INSERT INTO cart_items (owner_id, product_id, price, currency, quantity)
VALUES ($1, $2, $3, $4, $5)
ON CONFLICT (owner_id, product_id) DO UPDATE
    SET
        price = excluded.price, currency = excluded.currency,
        quantity = cart_items.quantity + excluded.quantity
RETURNING quantity;
```

Configuration in sqlc.yaml

```yaml
version: "2"
sql:
  - engine: "postgresql"
    database:
      uri: "postgres://postgres:password@localhost:5432/sqlc-deep-dive"
    schema: "internal/migrations"
    queries: "internal/db/queries"
    gen:
      go:
        package: "db"
        out: "internal/db"
        sql_package: "pgx/v5"
        emit_pointers_for_null_types: true
        overrides:
          - db_type: "uuid"
            go_type:
              import: "github.com/google/uuid"
              type: "UUID"
```

- sqlc generate



Generated code

```
  type AddItemParams struct {
    OwnerID   int64
    ProductID uuid.UUID
    Price     decimal.Decimal
    Currency  string
    Quantity  int32
}

func (q *Queries) AddItem(ctx context.Context, arg AddItemParams) (int32, error) {
    row := q.db.QueryRow(ctx, AddItem,
        arg.OwnerID, arg.ProductID, arg.Price, arg.Currency, arg.Quantity,
    )
    var quantity int32
    err := row.Scan(&quantity)
    return quantity, err
}
```

Generated code

```go
type AddItemParams struct {
    OwnerID   int64
    ProductID uuid.UUID
    Price     decimal.Decimal
    Currency  string
    Quantity  int32
}

func (q *Queries) AddItem(ctx context.Context, arg AddItemParams) (int32, error) {
    row := q.db.QueryRow(ctx, AddItem,
        arg.OwnerID, arg.ProductID, arg.Price, arg.Currency, arg.Quantity,
    )
    var quantity int32
    err := row.Scan(&quantity)
    return quantity, err
}
```

Three hard problems

1. Parameter type resolution

   What type is $1?

2. Output type resolution

   What type is each result column?

3. Query validation

   Is this query even valid?



Catalog



Queryable collection of **system** and **user-defined**:

- Tables (and views)
- Columns (name, type, nullability)
- Types (enums, domains)
- Functions (signatures + return types)

Simplified catalog

```go
package catalog

type Catalog struct {
    DefaultSchema string
    Schemas       []*Schema
}

type Schema struct {
    Name   string
    Tables []*Table
    Types  []Type
    Funcs  []*Function
}
```

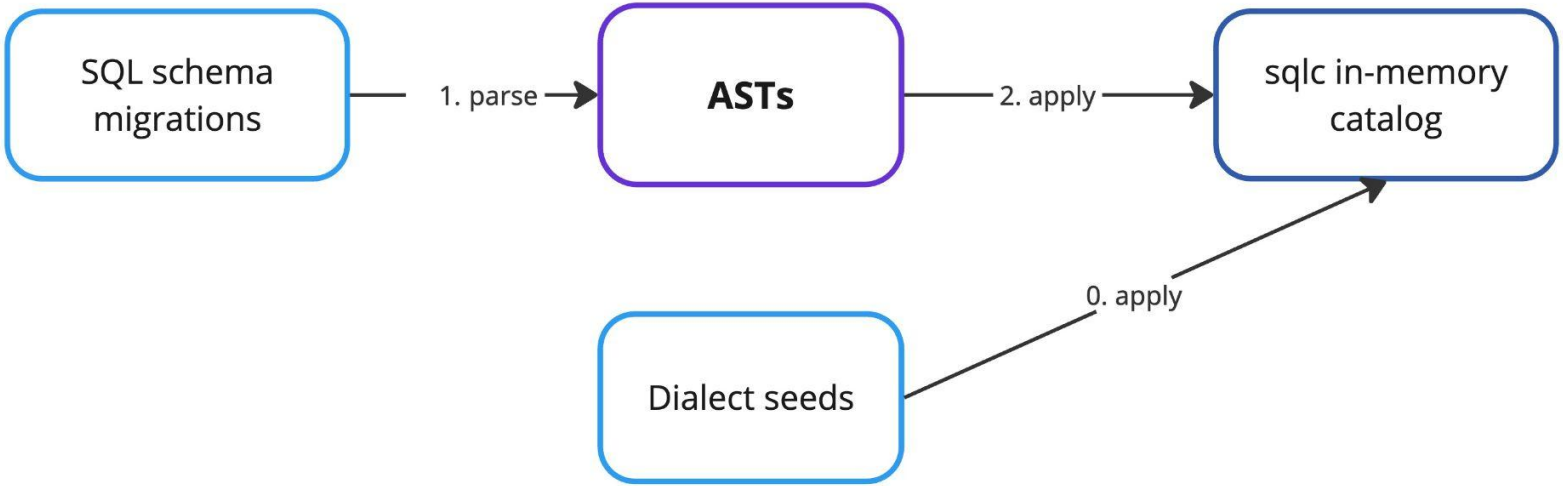
```go
type Table struct {
    Rel     *ast.TableName
    Columns []*Column
}

type Column struct {
    Name       string
    Type       ast.TypeName
    IsNotNull  bool
    IsUnsigned bool
    IsArray    bool
}
```

In-memory catalog

- Historically sqlc uses **in-memory** catalog only 😿
- DB-mode can refine **types** and nullability of output columns
- DB-mode for PostgreSQL was added only in 2023
- System catalog (*pg_catalog*, *information_schema*) is extracted from PG16, aka **dialect seeds**
  - relations, types, functions, operators
  - *goldeneye* utility, JSONL format

Building catalog



Dialect seeds example

```
relations.jsonl:
  {"schema":"pg_catalog","name":"pg_attribute","columns":[{"name":"attname",...}

types.jsonl:
  {"name": "int4", "category": "N", "aliases": ["integer", "int"]}

functions.jsonl:
  {"name":"now","returns":"timestamp with time zone"}

operators.jsonl:
  {"name": "+", "left": "date", "right": "int4", "result": "date"}
```

Abstract Syntax Tree



- Tree data structure
- Essential **structural** information of a SQL query
- *Abstract* - does NOT represent every single detail

AST example for CREATE TABLE

```
RawStmt
└── CreateTableStmt (if_not_exists: true)
    ├── name: TableName
    │   └── "cart_items"
    └── cols: []ColumnDef
        └── ColumnDef
            ├── colname: "owner_id"
            ├── is_not_null: true
            └── type_name: TypeName
                ├── schema: "pg_catalog"
                └── name: "int8"
```

Simplified *ast.CreateTableStmt*

```
type CreateTableStmt struct {
    Name        *TableName
    IfNotExists bool // true
    Cols        []*ColumnDef
}

type TableName struct {
    Catalog string // empty for PostgreSQL
    Schema  string // defaults to "public"
    Name    string // "cart_items"
}
```

Simplified *ast.ColumnDef*

```
type ColumnDef struct {
    Colname    string // "owner_id"
    TypeName   *TypeName
    IsNotNull  bool // true
    IsArray    bool // false
    PrimaryKey bool // false (for compound key)
}

type TypeName struct {
    Schema  string // "pg_catalog" for built-ins
    Name    string // "int8"
    Typmods *List  // nil; varchar(3), numeric(10,2)
}
```

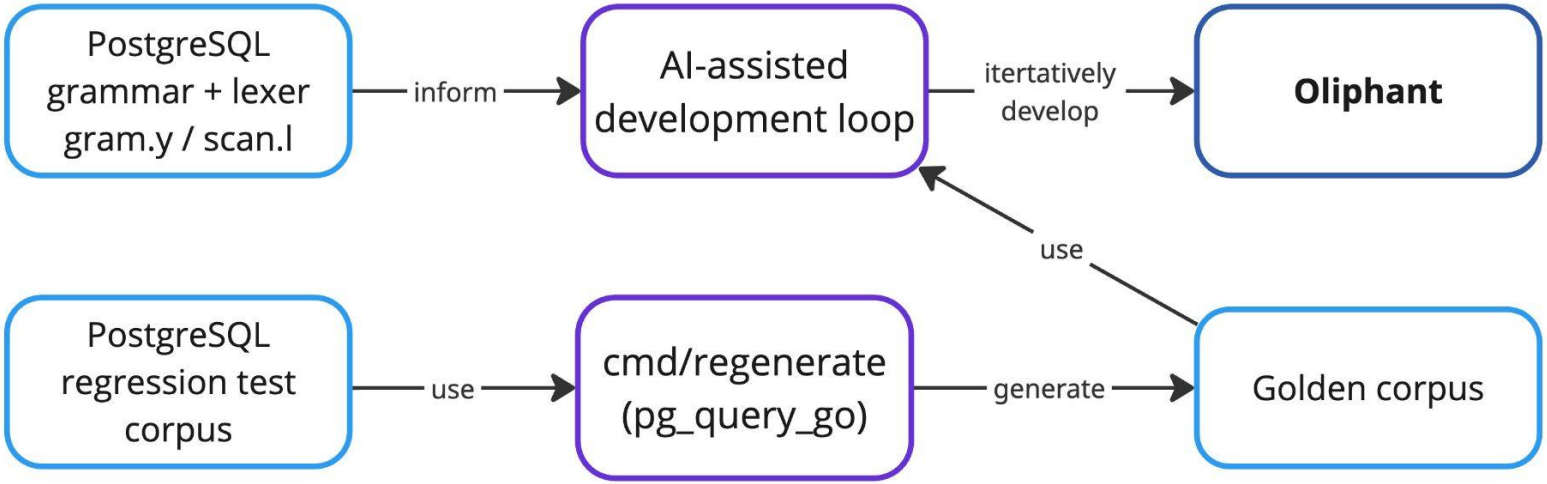
Libraries to parse AST

- *pganalyze/pg_query_go*
  - *libpg_query* - C-parser extracted from PostgreSQL
  - pg_query_go - CGO wrapper on top of *libpg_query*
  - Cons: no cross-compilation, C toolchain required, slow builds 😿
- *wasilibs/go-pgquery*
  - *libpg_query* compiled to WASM, *wazero* runtime
  - Cons: WASM overhead, lack of codebase control 😿
- **sqlc-dev/oliphant**

sqlc-dev/oliphant

- Pure Go from scratch; no CGO, no WASM
- Based on PostgreSQL Bison grammar file(s)
- Uses PostgreSQL regression test corpus
- Uses *pg_query_go* to produce **golden** test corpus
- Oliphant's ASTs can be compared against golden ASTs
- Corpus-driven TDD AI-assisted development loop

Building Oliphant



Oliphant AST to sqlc AST

- *libpg_query*, pg_query_go* and now ***oliphant*** expose structs generated from (the same) **Protobuf** schema
- The schema is PostgreSQL specific
- *sqlc* supports multiple engines: MySQL, SQLite, ClickHouse
- *sqlc* needs engine-agnostic AST representation and **conversion** 😢

Simplified *oliphant's ast.CreateStmt*

- *libpg_query*, pg_query_go* and ***oliphant*** expose Protobuf structs

```go
// Code generated by protoc-gen-go. DO NOT EDIT.
type CreateStmt struct {
    state          protoimpl.MessageState `protogen:"open.v1"`
    Relation       *RangeVar              `protobuf:"bytes,1,opt,name=relation,proto3"`
    TableElts      []*Node                `protobuf:"bytes,2,rep,name=table_elts,json=tableElts,proto3"`
    Constraints    []*Node                `protobuf:"bytes,7,rep,name=constraints,proto3"`
    IfNotExists    bool                   `protobuf:"varint,13,opt,name=if_not_exists,proto3"`
}
```

Oliphant AST to sqlc AST conversion

```
func translate(node *oliphant.Node) (ast.Node, error) {
    switch inner := node.Node.(type) {

    case *oliphant.Node_CreateStmt:
        create := &ast.CreateTableStmt{
            Name:        inner.CreateStmt.Relation.TableName(),
            IfNotExists: inner.CreateStmt.IfNotExists,
        }

        for _, elt := range inner.CreateStmt.TableElts {
          create.Cols = append(create.Cols, &ast.ColumnDef{...})
        }
```

Applying ASTs to catalog

```
func (c *Compiler) parseCatalog(files []schemaFile) {
    for _, file := range files {
        stmts := c.parser.Parse(file.contents)

        for i := range stmts {
            c.catalog.Update(stmts[i])
        }
    }
}

func (c *Catalog) Update(stmt ast.Statement) {
    switch n := stmt.Raw.Stmt.(type) {

    case *ast.CreateTableStmt:
        err = c.createTable(n)
```

Query analyzer



Analyzes each query (**AST**) and resolves:

- Parameters: types, names, nullability
- Output columns: types, names, nullability
- SELECT * into explicit columns

Extended Query Protocol

PostgreSQL can split query execution into discrete steps:

- **Parse** - parses and validates query, creates prepared statement
- **Describe** - returns metadata of the prepared statement
  - Parameters: type OIDs
  - Output columns: name, table OID, type OID, etc
- Bind, Execute, Sync, Close

OID = Object Identifier, unique for every object in the system catalog, e.g. text -> OID 25

Why not extended query protocol?

- Strong desire to support **in-memory** mode
- **Parameter names** extraction: *$1 → owner_id*
- **Query annotations** extraction:

  -- name: AddItem :one

- **Columns** expansion: *SELECT *, sqlc.embed()*

```
type AddItemParams struct {
    ProductID uuid.UUID
    Price     decimal.Decimal
    Currency  string
    Quantity  int32
    OwnerID   int64
}

type AddItemParams struct {
    arg1 uuid.UUID
    arg2 decimal.Decimal
    arg3 string
    arg4 int32
    arg5 int64
}
```

Query to analyze

```
-- name: AddItem :one
INSERT INTO cart_items (owner_id, product_id, price, currency, quantity)
VALUES (sqlc.arg(owner_id), $2, $3, $4, $5)
ON CONFLICT (owner_id, product_id) DO UPDATE
    SET
        price = excluded.price,
        currency = excluded.currency,
        quantity = cart_items.quantity + excluded.quantity
RETURNING quantity;
```

Preprocessor

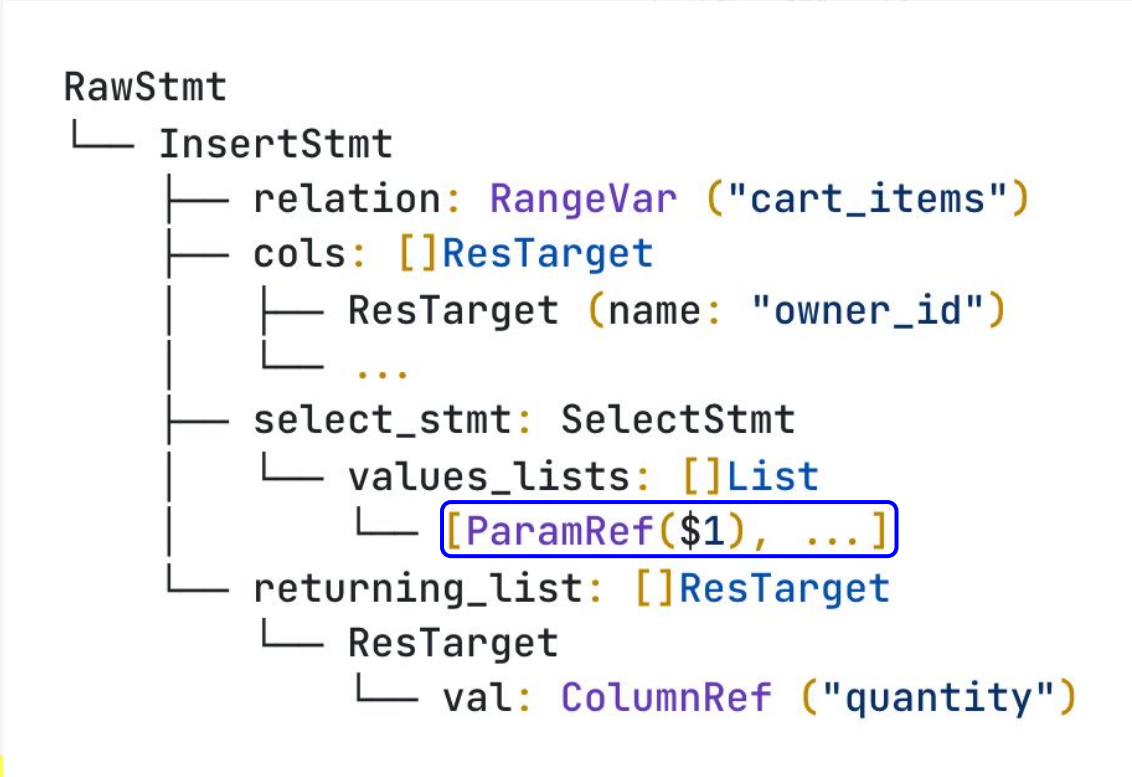
sqlc syntax has to be stripped before passing to *oliphant*

- Rewrites text and numbers placeholders
  - *sqlc.arg*, *sqlc.narg*, *sqlc.slice* => $1, $2, $3
- Stores metadata and byte offsets
  - AST might produce different parameters order

Preprocessed query to analyze

```
-- name: AddItem :one
INSERT INTO cart_items (owner_id, product_id, price, currency, quantity)
VALUES ($1, $2, $3, $4, $5)
ON CONFLICT (owner_id, product_id) DO UPDATE
    SET
        price = excluded.price, currency = excluded.currency,
        quantity = cart_items.quantity + excluded.quantity
RETURNING quantity;
```

- AST example for *INSERT INTO* (without conflict)



AST example for *INSERT INTO* (conflict clause)

```
InsertStmt
└── on_conflict_clause: OnConflictClause (DO UPDATE)
    ├── infer: InferClause
    │   └── index_elems: [IndexElem("owner_id"), IndexElem("product_id")]
    └── target_list: []ResTarget
        ├── ...
        └── ResTarget (name: "quantity")
            └── val: A_Expr ("+")
                ├── lexpr: ColumnRef ("cart_items.quantity")
                └── rexpr: ColumnRef ("excluded.quantity")
```

AST walk and visitor pattern

```go
package astutils

type Visitor interface {
    Visit(ast.Node) Visitor
}
```

```go
func Walk(f Visitor, node ast.Node) {
    if f = f.Visit(node); f == nil {
        return // stop descending
    }

    switch n := node.(type) {
    case *ast.InsertStmt:
        if n.Relation != nil {
            Walk(f, n.Relation)
        }
        // n.Cols, n.OnConflictClause,
        // n.SelectStmt, n.ReturningList

    // more node types
```

compiler.findParameters

- Walks query AST and collects every *ParamRef* ($1, $2)
- Collects context also - **parent** node and associated **table** (*RangeVar*)
- Custom visitor implementation - *compiler.paramSearch*
- Special handling for *InsertStmt*, *UpdateStmt*
  - column and parameter are in **separate** AST branches

compiler.findParameters

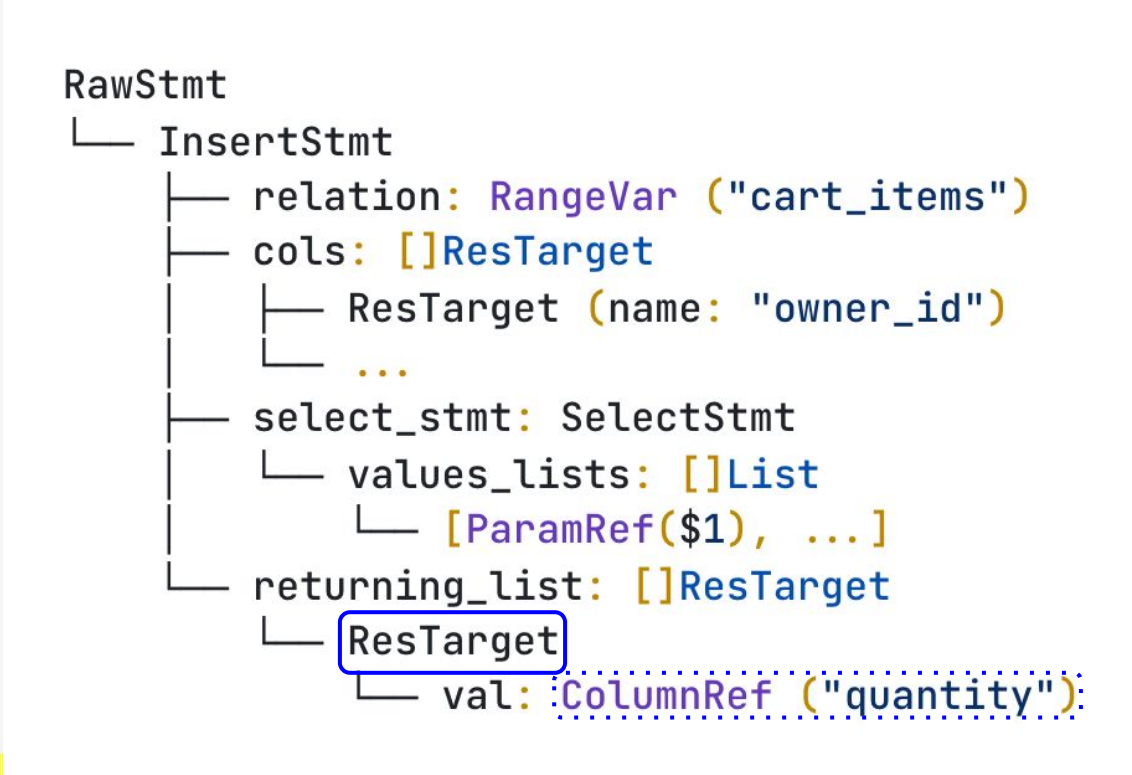
```
type paramRef struct {
    parent ast.Node
    rv     *ast.RangeVar // table/relation name
    ref    *ast.ParamRef // number($1) and location in SQL
}

func findParameters(root ast.Node) ([]paramRef, []error) {
    v := paramSearch{}
    astutils.Walk(v, root)
```

compiler.resolveCatalogRefs

- Converts []*paramRef* to []*Parameter* (typed, named, ready for code generation)
- Switches on ***ref.parent* node type** to know where **column name** in the AST
  - *A_Expr, BetweenExpr, ResTarget, FuncCall, TypeCast*
  - Fallbacks to *any*
- Looks up type and nullability by column name from catalog (*typeMap*)

AST example for *INSERT INTO - outputColumns*



compiler.outputColumns

- Iterates ***ResTarget*** nodes (instead of *ParamRef*) from known top-level fields
  - *TargetList* (*SELECT*), *ReturningList* (INSERT/UPDATE/DELETE)
- Supports ***CTEs*** resolution
- Star expansion (*SELECT* *)
- Not a generic AST walk, but *outputColumns* **recurses** into subqueries

Compiler result for code generation

```go
package compiler

type Query struct {
    SQL      string        // preprocessed and expanded
    Metadata Metadata      // parsed annotation comment
    Columns  []*Column     // analyzed output columns
    Params   []Parameter   // analyzed parameters
}

type Result struct {
    Queries []*Query

    // helpful when dealing with enum, custom types, type overrides
    Catalog *catalog.Catalog
}
```

Code generation

- Generator selection
- Intermediate Go representation
- Template rendering

Simplified query template

```
{{if .Arg.EmitStruct}}
type {{.Arg.Type}} struct { {{- range .Arg.Struct.Fields}}
  {{.Name}} {{.Type}} {{if .Tag}}{{$.Q}}{{.Tag}}{{$.Q}}{{end}}
  {{- end}}
}
{{end}}

{{if eq .Cmd ":one"}}
func (q *Queries) {{.MethodName}}(ctx context.Context, {{.Arg.Pair}}) ({{.Ret.DefineType}}, error) {
    row := q.db.QueryRow(ctx, {{.ConstantName}}, {{.Arg.Params}})
    {{- if or (ne .Arg.Pair .Ret.Pair) (ne .Arg.DefineType .Ret.DefineType) }}
    var {{.Ret.Name}} {{.Ret.Type}}
    {{- end}}
    err := row.Scan({{.Ret.Scan}})
    return {{.Ret.ReturnName}}, err
}
{{end}}
```

Rendered query

```
  type AddItemParams struct {
    OwnerID   int64
    ProductID uuid.UUID
    Price     decimal.Decimal
    Currency  string
    Quantity  int32
}

func (q *Queries) AddItem(ctx context.Context, arg AddItemParams) (int32, error) {
    row := q.db.QueryRow(ctx, AddItem,
        arg.OwnerID, arg.ProductID, arg.Price, arg.Currency, arg.Quantity,
    )
    var quantity int32
    err := row.Scan(&quantity)
    return quantity, err
}
```

Takeaways

- Brilliant and complex tool
- In-memory catalog
- Leverages *AST* and *oliphant*
- stdlib *text/template* rendering

Thank you!

Nikolay Kuznetsov

nkuznetsov

nikolayk812



nikolayk812.github.io/go-talks/20261103-golab/slides.pdf

