sqlc SQL to type-safe Go

START HERE

`sqlc init`
Create sqlc.yaml

`sqlc diff`
Is output stale?

`sqlc generate`
SQL to Go code

`sqlc verify`
Check against cloud tag

`sqlc compile`
Check SQL only

`-- name: <Name> :one`
Declare a query

`sqlc vet`
Lint queries

`sqlc.arg(<name>)`
Name a parameter

schema.sql tables or migrations → `sqlc generate` → **query.sql** named queries → `parsed and typed` → **Go package** models, methods → `you call` → **Your app** typed results

CONCEPTS

You write schema and queries in plain SQL; sqlc generates type-safe Go code, so the SQL is the source of truth.

Commit the generated code and run sqlc diff in CI to catch forgotten regeneration.

COMMANDS

`sqlc init`
Create an empty sqlc.yaml (v2 format)

`sqlc generate`
Generate code from schema and queries

`sqlc compile`
Check SQL for syntax and type errors
Parses and type-checks only; writes no files.

`sqlc vet`
Run lint rules over the queries

`sqlc diff`
Show how output differs from files on disk

`sqlc -f <file> generate`
Use another config file

`sqlc init --v1`
Create a version 1 config instead

`sqlc parse`
Print the SQL AST as JSON

`sqlc createdb`
Create an ephemeral database

`sqlc verify --against <tag>`
Compare to a tagged push

`sqlc push -t <tag>`
Upload project to sqlc cloud

`sqlc push --dry-run`
Dump the push request only

`sqlc --no-remote generate`
Disable remote execution

`sqlc version`
Print installed version

QUERY ANNOTATIONS

`-- name: <Name> :one`
Return a single row

`-- name: <Name> :many`
Return a slice of rows

`-- name: <Name> :exec`
Run it, return only an error

`-- name: <Name> :execrows`
Return affected row count

`-- name: <Name> :execresult`
Return the driver's sql.Result

`-- name: <Name> :execlastid`
Return the generated ID

`-- name: <Name> :batchexec`
Batch executions (pgx, Go only)

`-- name: <Name> :batchone`
Batch of single-row queries

`-- name: <Name> :batchmany`
Batch of multi-row queries

`-- name: <Name> :copyfrom`
Bulk insert, driver specific

PARAMETERS AND MACROS

`sqlc.arg(<name>)`
Name a parameter in the Go method

`sqlc.narg(<name>)`
Name a parameter and make it nullable
Useful for optional filters: WHERE x = coalesce(sqlc.narg(x), x).

`sqlc.slice("<name>")`
Slice in IN (...) for MySQL/ SQLite
Expanded at runtime, so not usable with prepared statements.

`sqlc.embed(<table>)`
Return the table's model struct as a field

Placeholders per engine

Engine	Placeholder
`postgresql`	$1, $2
`mysql`	?
`sqlite`	?

SQLC.YAML

`version: "2"`
Config format version

`engine: postgresql`
postgresql, mysql or sqlite

`schema: "<path>"`
Schema file, dir or migrations

`queries: "<path>"`
Query file or directory

`gen.go.package: <name>`
Go package name of the output

`gen.go.out: <dir>`
Output directory

`sql_package: pgx/v5`
Pick driver (default database/ sql)

`emit_json_tags: true`
Add json tags to structs

`emit_interface: true`
Generate a Querier interface

`emit_empty_slices: true`
Return [] instead of nil for :many

`emit_pointers_for_null_types`
Use *T for nullable columns

`overrides:`
Map db types or columns to Go types

`rules: [<name>]`
Enable vet rules for this package

`database.uri: ${DB_URL}`
Live database for vet; env vars expand

`database.managed: true`
Use a managed database (needs cloud)

`analyzer.database: false`
Skip database-backed analysis

`strict_function_checks: true`
Error on unknown functions

`codegen: / plugins:`
Plugin based output (process, wasm)

VET RULES

`sqlc/db-prepare`
Built-in: prepare every query on the DB
Needs database.uri; the DB must already have the current schema.

`-- @sqlc-vet-disable <rule>`
Skip a rule for one query

CEL variables in a rule

Variable	Holds
`query.sql`	SQL text
`query.name`	Query name
`query.cmd`	one, many, exec...
`query.params`	Parameter list
`config.engine`	Configured engine

RECIPES

Minimal sqlc.yaml

```
version: "2"
sql:
  - engine: "postgresql"
    schema: "schema.sql"
    queries: "query.sql"
    gen:
      go:
        package: "db"
        out: "db"
```

CRUD queries

```
-- name: GetAuthor :one
SELECT * FROM authors WHERE id =
$1;

-- name: ListAuthors :many
SELECT * FROM authors ORDER BY
name;

-- name: DeleteAuthor :exec
DELETE FROM authors WHERE id =
$1;
```

Custom vet rule

```
rules:
  - name: no-delete
    message: "don't use delete"
    rule:
query.sql.contains("DELETE")
# then add - no-delete under
sql[].rules
```

CI freshness check

```
sqlc compile
sqlc vet
sqlc diff   # non-zero if
generated code is stale
```