pggen: Go code generation for Postgres

Working example: https://github.com/jschaf/pggen/tree/master/example/author

pggen is a binary that generates Go code that provides a typesafe wrapper to Postgres queries. pggen has the same goals as sqlc, to "compile SQL to type-safe Go". The sqlc documentation provides a concise overview of the benefits of the code generation approach:

> *sqlc generates fully-type safe idiomatic Go code from SQL.*
>
> - *You write SQL queries*
> - *You run sqlc to generate Go code that presents type-safe interfaces to those queries*
> - *You write application code that calls the methods sqlc generated.*

The primary difference between pggen and sql**c** is how pggen generates the Go code. sqlc parses the queries in Go code, using Cgo to call the Postgres parser.c code. After parsing, sqlc infers the types of the query parameters and result columns using custom logic in Go. In contrast, pggen gets the same type information by running the queries on Postgres and then fetching the type information from the Postgres catalog tables. Here are the merits and downsides of both approaches:

	sqlc	**pggen**
Pros	Standalone and lightweight. Doesn't require running a Postgres instance or dealing with Docker.	Perfect type information because Postgres reports the types by executing the actual query. pggen makes no assumptions about the version so end users can "bring their own database" with any schema and extensions. Easy to maintain. Postgres does all the hard work of getting type information so pggen doesn't need to reimplement the Postgres type inference semantics.
Cons	Type inference is buggy. sqlc fails on a variety of queries (CTEs, UNION, all Postgres bugs). Difficult to maintain because sqlc needs to match the exact semantics of Postgres type inference. Postgres is a	Requires running queries on a user-provided Postgres cluster. This makes integration more complex compared to sqlc. To reduce the integration complexity, pggen could provide an up-to-date Postgres runner through Docker.

	moving target that constantly evolves. Requires vendoring all Postgres function and extension types. As a consequence, sqlc only supports one version of Postgres, or a superset of many versions.	Running queries is potentially destructive, though the queries should only ever run on temporary databases purposefully created only for pggen.

I believe pggen is much simpler to both implement and maintain compared to sqlc. Trying to implement Postgres type inference semantics in Go is both a herculean (see sqlc's parser.go) and Sisyphean task as Postgres is a moving target. By leveraging Postgres directly to provide type information, pggen will always have perfect type information with no extra effort in the implementation.

Secondary goals

- Support pre and post query execution hooks for cross-cutting concerns like logging and tracing.

Non-goals

- pggen will not support the Go standard sql interface. pggen is built on top of pgx, and exposes advanced pgx features like binary wire format, custom Postgres types, and query batching.

Implementation sketch

The core task for pggen is to determine the input and output types for an arbitrary query, given a user-provided Postgres database.

Determine query input types

Determining the query input types is straightforward. The steps are:

1. Prepare a query.
2. Query the system catalog table, pg_prepared_statements, to get an array of parameter types for the prepared query.

As an example, we'll use a prepared query with three parameters with different types. First, prepare the query:

```
PREPARE sample_input_query AS
```

```
SELECT book_id, $1::text AS text_col
FROM books
WHERE book_id = $2
  AND $3 < '2020-01-04'::date;
```

Second, query `pg_prepared_statements` for the parameter types:

```
SELECT parameter_types
FROM pg_prepared_statements
WHERE name = 'sample_input_query';
```

The query returns: `{text,integer,date}` which are the correct types for the three query parameters.

Determine query output types

Determining the query output types is a bit tricky. Stackoverflow was helpful in finding a solution. The steps are:

1. Determine if the query has any output. Mutation queries without a `RETURNING` clause always return sql.Result.
   Run the query using `EXPLAIN (VERBOSE, FORMAT JSON)`. If `Plan[Node Type]` is `Modify Table`, meaning an insert, update, or delete statement, only continue if `Plan[Output]` is defined which means the statement has a `RETURNING` clause. Otherwise, the query has no output.
2. Prepare the query.
3. Create a temp table by executing the prepared query using a default value or null for each parameter.
4. Query the `pg_attribute` table to get the type information for each column in the temp table. The columns of the temp table match the output columns of the query.

As an example, we'll use a query with moderately complex types that takes a single parameter. First, prepare the query.

```
PREPARE sample_query AS
  SELECT book_id,
         book_type,
         'arbitrary_column' as text_col,
         '2021-01-14'::date - INTERVAL '3 hour' as book_time
  FROM books
  WHERE book_id = $1;
```

Second, create a temp table using the query:

```
CREATE TEMP TABLE tmp_sample AS
  EXECUTE sample_query ( NULL );
```

Third, get the types from the pg_attribute table for the temp table:

```
SELECT attname, format_type(atttypid, atttypmod) AS type
FROM pg_attribute
WHERE attrelid = 'tmp_sample'::regclass
  AND attnum > 0
  AND NOT attisdropped
ORDER BY attnum;
```

This pg_attribute query returns a table of the query result column names and their types.

attname	type
book_id	integer
book_type	book_type
text_col	text
book_time	timestamp without time zone

Unresolved questions

Executing insert and update statements with a returning clause can violate check, null, or unique constraints

pggen runs queries as prepared queries using null for each parameter. This approach (always?) works for select-statements but can fail for update and insert queries by violating not-null constraints, check constraints, or unique constraints on the target table. The problem is limited in scope because pggen only needs to execute statements that contain a `RETURNING` clause to get the output types. Without a returning clause, modification queries always return a command tag type that contains the number of rows modified, so pggen doesn't need to execute the query.

One approach is to require the user to specify a default value in pggen.arg as a second argument, like `pggen.arg('user_id', 876)`. As a convenience feature, pggen could use default values for all known types. Providing a default value avoids violating not-null constraint and if carefully chosen, also avoids check constraint and unique constraint.

Another partial solution is to constrain the query so that it doesn't do anything by adding a `LIMIT 0` clause or with a `WHERE false` clause. This will likely work for the vast majority of queries but it's always possible for this to fail in complex SQL queries. A disadvantage of this approach is that it requires parsing the query and then manipulating the parse tree which is a large bundle of added complexity. Parsing the query in application code also means pggen is tied to a specific Postgres version for query parsing semantics. pggen will likely need to parse statements anyway so this might be a reasonable tradeoff.

API Design

The API design includes both the SQL metadata format to identify queries and the generated Go output code.

SQL metadata format

pggen both simplifies and extends the sqlc format described in sqlc annotations doc:

> *sqlc requires each query to have a small comment indicating the name and command. The format of this comment is as follows:*

```
-- name: <name> <command>
-- name: GetAuthor :one
-- name: FindAuthors :many
-- name: DeleteAuthor :exec
```

Simplification: only offer exec command which returns pgconn.CommandTag

Running a modification query returns two things, sql.Result and any error from running the query. sqlc provides :exec which only returns the error, :execresult which returns both sql.Result and the error, and :execrows which returns the sql.Result.RowsAffected() and the error.. Both exec and execrows are minor convenience wrappers around sql.Result.

pggen slightly simplifies allowed annotations by only supporting :exec. Instead of returning sql.Result, pggen returns pgconn.CommandTag which is the Postgres equivalent of sql.Result.

Extension: reuse result types among different queries

As an extension to sqlc, pggen provides the ability to explicitly set the output type. sqlc creates a new struct for each query unless the query is all rows in the table. Oftentimes, queries differ only by a where-clause but return the same set of columns. Having a different type for each query is cumbersome since it requires manual translation to a unified struct. Instead, pggen

supports an optional SQL annotation on select statements to specify the name of the struct representing a row in the query results.

```
-- name: FindAuthorNames :many :resultName=AuthorName
```

Other queries may reuse the result type (AuthorName) of FindAuthorNames like so:

```
-- name: FindBestSellingAuthors :many :resultFromQuery=FindAuthorNames
```

A query with a `:resultFromQuery` annotation must meet the following criteria:

- The columns of the query must be a subset (including a complete subset) of the columns of the referenced query. If FindAuthorNames returns `{last_name: text, id: bigint}`, FindBestSellingAuthors can return any subset: `{id: bigint}`, `{last_name: text}`, or `{last_name: text, id: bigint}`.
- Each result column must have the exact same types as the reference query.

Simplification: all query parameters must be named

sqlc allows both named and anonymous parameters but pggen only supports named parameters in order to simplify the implementation.

```
-- name: GetRecord :one
SELECT * FROM records WHERE id = $1;

-- name: GetRecord :one
SELECT * FROM records WHERE id = sqlc.arg('id');
```

This limitation simplifies the implementation logic especially when multiple queries share the same result type. With anonymous parameters, pggen would either need to match the order of types or the name of the types. Both approaches are ambiguous and error prone. Instead pggen only supports named parameters by providing marker function, `pggen.arg`, for parameter names. `pggen.arg` takes a text parameter, the name of the column and an optional named parameter, `test_val`, to specify the value to use when executing the query to infer the types.

The purpose of the test_val argument is to allow insert and update statements to pass check constraints when modifying tables. test_val does nothing for select statements.

```
-- name: GetRecord :one
SELECT * FROM records WHERE id = pggen.arg('id', test_val := 'some_id');
```

Generated Go code

The generated code will use pgx and mimic the sqlc output. Since pggen is a Postgres specific code generator, there's limited benefit to using Go's standard SQL types. Each pggen config file will generate:

- A Querier interface containing all of the SQL queries in the file querier.go.
- A DBQuerier struct that implements the Querier interface, also in querier.go.
- For each sql query file, a generated file name: basename(sqlFile).go

Postgres client tracing

To support tracing, the package pggen provides ClientTrace (trace.go), similar to httptrace.ClientTrace.