cql

Franco Liberali

Jan 15, 2024

GENERAL

CHAPTER

ONE

WHAT IS CQL?

Originally part of BaDaaS, CQL allows easy and safe persistence and querying of objects.

It's built on top of gorm, a library that actually provides the functionality of an ORM: mapping objects to tables in the SQL database. While gorm does this job well with its automatic migration then performing queries on these objects is somewhat limited, forcing us to write SQL queries directly when they are complex. CQL seeks to address these limitations with a query system that:

- Is compile-time safe: queries are validated at compile time to avoid errors such as comparing attributes that are of different types, trying to use attributes or navigate relationships that do not exist, using information from tables that are not included in the query, etc.; ensuring that a runtime error will not be raised.
- Is easy to use: the use of its query system does not require knowledge of databases, SQL languages or complex concepts. Writing queries only requires programming in Go and the result is easy to read.
- Is designed for real applications: the query system is designed to work well in real-world cases where queries are complex, require navigating multiple relationships, performing multiple comparisons, etc.
- Is designed so that developers can focus on the business model: its queries allow easy retrieval of model relationships to apply business logic to the model and it provides mechanisms to avoid errors in the business logic due to mistakes in loading information from the database.
- It is designed for high performance: the query system avoids as much as possible the use of reflection and aims that all the necessary model data can be retrieved in a single query to the database.

Language	Query
SQL	SELECT cities.* FROM cities INNER JOIN countries ON countries.id = cities.country_id AND countries.name = "France" WHERE cities.name = "Paris"
GORM	db.Where("cities.name = ?","Paris",).Joins("Country",db.Where("Country.name = ?", "France",),).Find(&cities)
CQL	cql.Query[models.City](db, conditions.City.Name.Is().Eq("Paris"), conditions.City.Country(conditions.Country.Name.Is().Eq("France"),),).FindOne()

CHAPTER

TWO

IS CQL A COPY OF GORM-GEN?

It is true that its aim seems to be the same:

> 100% Type-safe DAO API without interface{}

Although gorm-gen provides a more structured API than gorm for performing queries, providing methods like:

```
Where(conds ...gen.Condition) IUserDo
```

we can see from this signatures that, for example, the Where method receives parameters of type gen.Condition. In this way, conditions from different models could be mixed without generating a compilation error:

```
u := query.User
c := query.Company
user, err := u.Where(c.Name.Eq("franco")).First()
```

which would generate a runtime error during the execution of the generated SQL:

```
SELECT * FROM `users` WHERE `companies`.`name` = "franco"
no such column: companies.name
```

Because of this, cql decides to go further in type safety and check that the conditions are of the correct model, that the compared values are of the same type, that the models are included in the query and more, ensuring that a runtime error will not be raised.

For more details see: *Type safety*.

CHAPTER

THREE

QUICKSTART

To integrate cql into your project, you can head to the quickstart.

3.1 Run it

Refer to its README.md for running it.

3.2 Understand it

Once you have started your project with *go init*, you must add the dependency to cql:

```
go get -u github.com/FrancoLiberali/cql gorm.io/gorm
```

Create a package for your *models*, for example:

```
package models

import (
  "github.com/FrancoLiberali/cql/model"
)

type MyModel struct {
  model.UUIDModel

  Name string
}
```

Once done, you can *generate the conditions* to perform queries on them. In this case, the file *conditions/cql.go* has the following content:

```
package conditions

//go:generate cql-gen ../models
```

Then, you can generate the conditions running:

```
go generate ./...
```

In *main.go* there is a main function that creates a *gorm.DB* that allows connection with the database and calls the *AutoMigrate* method with the models you want to be persisted.

After this, you are ready to query your objects using cql.Query.

Now that you know how to integrate cql into your project, you can learn how to use it by following the *Tutorial*.

CHAPTER

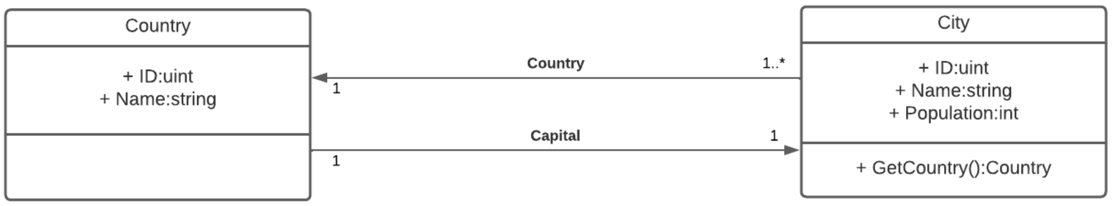
FOUR

TUTORIAL

In this short tutorial you will learn the main functionalities of cql. The code to be executed in each step can be found in this repository.

4.1 Model and data

In the file *models/models.go* you find the definition of the following model:



For details about the definition of models you can read *Declaring models*.

In *sqlite:db* you will find a sqlite database with the following data:

Table 1: Countries

ID	Name	CapitalID
1	United States of America	2
2	France	3

Table 2: Cities

ID	Name	Population	CountryID
1	Paris	25171	1
2	Washington D. C.	689545	1
3	Paris	2161000	2

As you can see, there are two cities called Paris in this database: the well known Paris, capital of France and site of the iconic Eiffel tower, and Paris in the United States of America, site of the Eiffel tower with the cowboy hat (no joke, just search for paris texas eiffel tower in your favorite search engine).

In this tutorial we will explore the cql functions that will allow us to differentiate these two Paris.

4.2 Tutorial 1: simple query

In this first tutorial we are going to perform a simple query to obtain all the cities called Paris.

In the tutorial_1.go file you will find that we can perform this query as follows:

```
cities, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
).Find()
```

We can run this tutorial with *make tutorial_1* and we will obtain the following result:

```
Cities named 'Paris' are:
    1: City{ID: 1, Name: Paris, Population: 25171, CountryID:1, Country:<nil> }
    2: City{ID: 3, Name: Paris, Population: 2161000, CountryID:2, Country:<nil> }
```

As you can see, in this case we will get both cities which we can differentiate by their population and the id of the country.

In this tutorial we have used the cql compiled queries system to get these cities, for more details you can read *Conditions*.

4.3 Tutorial 2: operators

Now we are going to try to obtain only the Paris of France and in a first approximation we could do it using its population: we will only look for the Paris whose population is greater than one million inhabitants.

In the tutorial_2.go file you will find that we can perform this query as follows:

```
cities, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
    conditions.City.Population.Is().Gt(1000000),
).Find()
```

We can run this tutorial with *make tutorial_2* and we will obtain the following result:

```
Cities named 'Paris' with a population bigger than 1.000.000 are:
    1: City{ID: 3, Name: Paris, Population: 2161000, CountryID:2, Country:<nil> }
```

As you can see, in this case we only get one city, Paris in France.

In this tutorial we have used the operator Gt to obtain this city, for more details you can read *Operators*.

4.4 Tutorial 3: modifiers

Although in the previous tutorial we achieved our goal of differentiating the two Paris, the way to do it is debatable since the population of Paris, Texas may increase to over 1000000 someday and then, the result of this query can change. Therefore, we will search only for the city with the largest population.

In the tutorial_3.go file you will find that we can perform this query as follows:

```
parisFrance, err := cql.Query[models.City](
            db,
            conditions.City.Name.Is().Eq("Paris"),
    ).Descending(
            conditions.City.Population,
    ).Limit(1).FindOne()
```

We can run this tutorial with *make tutorial_3* and we will obtain the following result:

```
City named 'Paris' with the largest population is: City{ID: 3, Name: Paris, Population:␣
↪2161000, CountryID:2, Country:<nil> }
```

As you can see, again we get only the Paris in France. As you may have noticed, in this case we have used the *FindOne* method instead of *Find*. This is because in this case we are sure that the result is a single model, so instead of getting a list we get a single city.

In this tutorial we have used query modifier methods, for more details you can read *Query methods*.

4.5 Tutorial 4: joins

Again, the solution of the previous tutorial is debatable because the evolution of populations could make Paris, Texas have more inhabitants than Paris, France one day. Therefore, we are now going to improve this query by obtaining the city called Paris whose country is called France.

In the tutorial_4.go file you will find that we can perform this query as follows:

```
parisFrance, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
    conditions.City.Country(
        conditions.Country.Name.Is().Eq("France"),
    ),
).FindOne()
```

We can run this tutorial with *make tutorial_4* and we will obtain the following result:

```
City named 'Paris' in 'France' is: City{ID: 3, Name: Paris, Population: 2161000,␣
↪CountryID:2, Country:<nil> }
```

As you can see, again we get only the Paris in France.

In this tutorial we have used a condition that performs a join.

4.6 Tutorial 5: preloading

You may have noticed that in the results of the previous tutorials the Country field of the cities was null (Country:<nil>). This is because, to ensure performance, cql will retrieve only the attributes of the model you are querying (City in this case because the method used is cql.Query[models.City]) but not of its relationships. If we also want to obtain this data, we must perform preloading.

In the tutorial_5.go file you will find that we can perform this query as follows:

```
cities, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
    conditions.City.Country().Preload(),
).Find()
```

We can run this tutorial with *make tutorial_5* and we will obtain the following result:

```
Cities named 'Paris' are:
    1: City{ID: 1, Name: Paris, Population: 25171, CountryID:1, Country:Country{ID: 1,␣
↪Name: United States of America, CapitalID:2, Capital:<nil> } } with country: Country
↪{ID: 1, Name: United States of America, CapitalID:2, Capital:<nil> }
    2: City{ID: 3, Name: Paris, Population: 2161000, CountryID:2, Country:Country{ID: 2,␣
↪Name: France, CapitalID:3, Capital:<nil> } } with country: Country{ID: 2, Name: France,
↪ CapitalID:3, Capital:<nil> }
```

As you can see, now the country attribute is a valid pointer to a Country object (Country:0xc0001d1600). Then the Country object information is accessed with the *GetCountry* method. This method is not defined in the *models/models.go* file but is a *relation getter* that is generated by cql-gen together with the conditions. These methods allow us to differentiate null objects from objects not loaded from the database, since when trying to browse a relation that was not loaded we will get *cql.ErrRelationNotLoaded*.

In this tutorial we have used preloading and relation getters, for more details you can read *Preloading*.

4.7 Tutorial 6: dynamic operators

So far we have performed operations that take as input a static value (equal to “Paris” or greater than 1000000) but what if now we would like to differentiate these two Paris from each other based on whether they are the capital of their country.

In the tutorial_6.go file you will find that we can perform this query as follows:

```
cities, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
    conditions.City.Country(
        conditions.Country.CapitalID.IsDynamic().Eq(conditions.City.ID.Value()),
    ),
).Find()
```

We can run this tutorial with *make tutorial_6* and we will obtain the following result:

```
Cities named 'Paris' that are the capital of their country are:
    1: City{ID: 3, Name: Paris, Population: 2161000, CountryID:2, Country:<nil> }
```

As you can see, again we only get the Paris in France.

In this tutorial we have used dynamic conditions, for more details you can read *Dynamic operators*.

4.8 Tutorial 7: update

So far we have only made select queries, but in this tutorial we want to edit the population of Paris.

In the tutorial_7.go file you will find that we can perform this query as follows:

```
updated, err := cql.Update[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
    conditions.City.Country(
        conditions.Country.Name.Is().Eq("France"),
    ),
).Returning(&cities).Set(
    conditions.City.Population.Set().Eq(2102650),
)
```

We can run this tutorial with *make tutorial_7* and we will obtain the following result:

```
Updated 1 city: City{ID: 3, Name: Paris, Population: 2102650, CountryID:2, Country:<nil>␣
↪}
Initial population was 2161000
```

As you can see, first we can know the number of updated models with the value "updated" returned by the Set method (according to the number of models that meet the conditions entered in the Update method). On the other hand, it is also possible to obtain the information of the updated models using the Returning method.

In this tutorial we have used updates, for more details you can read *Update*.

4.9 Tutorial 8: create and delete

In this tutorial we want to create a new city called Rennes and then delete it.

In the tutorial_8.go file you will find that we can perform this query as follows:

Listing 1: Create

```
rennes := models.City{
    Country:    france,
    Name:       "Rennes",
    Population: 215366,
}
if err := db.Create(&rennes).Error; err != nil {
    log.Panicln(err)
}
```

Listing 2: Delete

```
deleted, err := cql.Delete[models.City](
    db,
    conditions.City.Name.Is().Eq("Rennes"),
).Exec()
```

We can run this tutorial with *make tutorial_8* and we will obtain the following result:

```
Deleted 1 city
```

Here, we simply get the number of deleted models through the "deleted" variable returned by the Exec method (according to the number of models that meet the conditions entered in the Delete method).

In this tutorial we have used create and delete, for more details you can read *Create* and *Delete*.

4.10 Tutorial 9: Collections

In this tutorial we want to obtain all the countries that have a city called 'Paris'

In the tutorial_9.go file you will find that we can perform a query as follows:

```
countries, err := cql.Query[models.Country](
    db,
    conditions.Country.Cities.Any(
        conditions.City.Name.Is().Eq("Paris"),
    ),
).Find()
```

We can run this tutorial with *make tutorial_9* and we will obtain the following result:

```
Countries that have a city called 'Paris' are:
    1: Country{ID: 1, Name: United States of America, CapitalID:2, Capital:<nil> }
    2: Country{ID: 2, Name: France, CapitalID:3, Capital:<nil> }
```

As you can see, again we only get the Paris in France.

In this tutorial we have used conditions over collections, for more details you can read *Collections*.

4.11 Tutorial 10: Compile type safety

In this tutorial we want to verify that cql is compile-time safe.

In the tutorial_10.go file you will find that we try to perform a query as follows:

```
_, err := cql.Query[models.City](
    db,
    conditions.Country.Name.Is().Eq("Paris"),
).Find()
```

We can run this tutorial with *make tutorial_10* and we will obtain the following error during compilation:

```
./tutorial_10.go:20:3:
    cannot use conditions.Country.Name.Is().Eq("Paris")
    (value of type condition.WhereCondition[models.Country]) as condition.
↪Condition[models.City]...
```

As you can see, in this tutorial we are trying to put a condition on Country (conditions.Country) to a Query whose main model is City (Query[models.City]). This would be equivalent to trying to execute the following SQL query:

```
SELECT * FROM cities
WHERE countries.name = "Paris"
```

Therefore, we will get a compilation error and this incorrect code will never be executed.

For more details you can read *Type safety*.

CHAPTER

FIVE

CONCEPTS

5.1 Model

A model is any object (struct) of go that you want to persist in the database and on which you can perform queries. For this, the struct must have an embedded cql base model.

For details visit *Model declaration*.

5.2 Base model

It is a struct that when embedded allows your structures to become cql models, adding ID, CreatedAt, UpdatedAt and DeletedAt attributes and the possibility to persist, create conditions and perform queries on these structures.

For details visit *Base models*.

5.3 Model ID

The id is a unique identifier needed to persist a model in the database. It can be a model.UIntID or a model.UUID, depending on the base model used.

For details visit *Base models*.

5.4 Auto Migration

To persist the models it is necessary to migrate the database, so that the structure of the tables corresponds to the definition of the model. This migration is performed by gorm through the gormDB.

For details visit *Migration*.

5.5 GormDB

GormDB is a gorm.DB object that allows communication with the database. This object will be needed as a parameter for the main cql functions (Query, Update and Delete).

For details visit *Connection*.

5.6 Condition

Conditions are the basis of the cql query system, every query is composed of a set of conditions. Conditions belong to a particular model and there are 4 different types: WhereConditions, ConnectionConditions and JoinConditions.

For details visit *Query*.

5.7 WhereCondition

Type of condition that allows filters to be made on the model to which they belong and an attribute of this model. These filters are performed through operators.

For details visit *Query*.

5.8 ConnectionCondition

Type of condition that allows the use of logical operators (and, or, or, not) between WhereConditions.

For details visit *Query*.

5.9 JoinCondition

Condition type that allows to navigate relationships between models, which will result in a join in the executed query (don't worry, if you don't know what a join is, you don't need to understand the queries that cql executes).

For details visit *Query*.

5.10 Operator

Concept similar to database operators, which allow different operations to be performed on an attribute of a model, such as comparisons, predicates, pattern matching, etc.

Operators can be classified as static, dynamic and unsafe.

For details visit *Query*.

5.11 Static operator

Static operators are those that perform operations on an attribute and static values, such as a boolean value, an integer, etc.

For details visit *Query*.

5.12 Dynamic operator

Dynamic operators are those that perform operations between an attribute and other attributes, either from the same model or from a different model, as long as the type of these attributes is the same.

For details visit *Advanced query*.

5.13 Unsafe operator

Unsafe operators are those that can perform operations between an attribute and any type of value or attribute.

For details visit *Advanced query*.

5.14 Nullable types

Nullable types are the types provided by the sql library that are a nullable version of the basic types: sql.NullString, sql.NullTime, sql.NullInt64, sql.NullInt32, sql.NullBool, sql.NullFloat64, etc..

For details visit <https://pkg.go.dev/database/sql>.

5.15 Compiled query system

The set of conditions that are received by the *cql.Query*, *cql.Update* and *cql.Delete* methods form the cql compiled query system. It is so named because the conditions will verify at compile time that the query to be executed is correct.

For details visit *Conditions* and *Type safety*.

5.16 Conditions generation

Conditions are the basis of the compiled query system. They are generated for each model and attribute and can then be used. Their generation is done with cql-gen.

For details visit *Conditions generation*.

5.17 Relation getter

Relationships between objects can be loaded from the database using the Preload method. In order to safely navigate the relations in the loaded model cql provides methods called “relation getters”.

For details visit *Preloading*.

CHAPTER

SIX

DECLARING MODELS

6.1 Model declaration

The cql *model* declaration is based on the GORM model declaration, so its definition, conventions, tags and associations are compatible with cql. For details see gorm documentation. On the contrary, cql presents some differences/extras that are explained in this section.

6.2 Base models

To be considered a model, your structures must have embedded one of the *base models* provided by cql:

- *model.UUIDModel*: Model identified by a model.UUID (Random (Version 4) UUID).
- *model.UIntModel*: Model identified by a model.UIntID (auto-incremental uint).

Both base models provide date created, updated and deleted.

To use them, simply embed the desired model in any of your structs:

```
type MyModel struct {
  model.UUIDModel

  Name         string
  Email        *string
  Age          uint8
  Birthday     *time.Time
  MemberNumber sql.NullString
  ActivatedAt  sql.NullTime
  // ...
}
```

6.3 Type of attributes

As we can see in the example in the previous section, the attributes of your models can be of multiple types, such as basic go types, pointers, and *nullable types*.

This difference can generate differences in the data that is stored in the database, since saving a model created as follows:

```
MyModel{}
```

will save a empty string for Name but a NULL for the Email and the MemberNumber.

The use of nullable types is strongly recommended and cql takes into account their use in each of its functionalities.

6.4 Associations

All associations provided by GORM are supported. For more information see <https://gorm.io/docs/belongs_to.html>, <https://gorm.io/docs/has_one.html>, <https://gorm.io/docs/has_many.html> and <https://gorm.io/docs/many_to_many.html>. However, in this section we will give some differences in cql and details that are not clear in this documentation.

6.4.1 IDs

Since cql base models use model.UUID or model.UIntID to identify the models, the type of id used in a reference to another model is the corresponding one of these two, for example:

```
type ModelWithUUID struct {
  model.UUIDModel
}

type ModelWithUIntID struct {
  model.UIntModel
}

type ModelWithReferences struct {
  model.UUIDModel

  ModelWithUUID *ModelWithUUID
  ModelWithUUIDID *model.UUID

  ModelWithUIntID *ModelWithUIntID
  ModelWithUIntIDID *model.UIntID
}
```

6.4.2 References

References to other models can be made with or without pointers:

```go
type ReferencedModel struct {
  model.UUIDModel
}

type ModelWithPointer struct {
  model.UUIDModel

  // reference with pointer
  PointerReference *ReferencedModel
  PointerReferenceID *model.UUID
}

type ModelWithoutPointer struct {
  model.UUIDModel

  // reference without pointer
  Reference ReferencedModel
  ReferenceID model.UUID
}
```

As in the case of attributes, this can make a difference when persisting, since one created as follows:

```go
ModelWithoutPointer{}
```

will also create and save an empty ReferencedModel{}, what may be undesired behavior. For this reason, although both options are still compatible with cql, we recommend the use of pointers for references. In case the relation is not nullable, use the *not null* tag in the id of the reference, for example:

```go
type ReferencedModel struct {
  model.UUIDModel
}

type ModelWithPointer struct {
  model.UUIDModel

  // reference with pointer not null
  PointerReference *ReferencedModel
  PointerReferenceID *model.UUID `gorm:"not null"`
}
```

6.5 Reverse reference

Although no example within the gorm's documentation shows it, when defining relations, we can also put a reference in the reverse direction to add navigability to our model. In addition, adding this reverse reference will allow the corresponding conditions to be generated during condition generation.

For example:

```go
type Related struct {
  model.UUIDModel

  YourModel *YourModel
}

type YourModel struct {
  model.UUIDModel

  Related *Related
  RelatedID *model.UUID
}
```

CHAPTER

SEVEN

CONNECTING TO A DATABASE

7.1 Connection

cql supports the databases MySQL, PostgreSQL, SQLite, SQL Server using gorm's driver. Some databases may be compatible with the mysql or postgres dialect, in which case you could just use the dialect for those databases (from which CockroachDB is tested).

To communicate with the database, cql needs a *GormDB* object. To create it, you can use the function *cql.Open* that will allow you to connect to a database using the specified dialector. This function is equivalent to *gorm.Open* but with the difference that in case of not adding any configuration, the cql default logger will be configured instead of the gorm one. For details about this logger visit *Logger*. For details about gorm configuration visit gorm documentation.

7.2 Migration

Migration is done by gorm using the *gormDB.AutoMigrate* method. For details visit gorm docs.

CHAPTER EIGHT

TYPE SAFETY

8.1 Compile time safety

One of the most important features of the CQL is

```
Is compile-time safe:
    queries are validated at compile time to avoid errors
    such as comparing attributes that are of different types,
    trying to use attributes or navigate relationships that do not exist,
    using information from tables that are not included in the query, etc.;
    ensuring that a runtime error will not be raised.
```

While there are other libraries that provide an API type safety (gorm-gen, jooq (Java), diesel (Rust)), CQL is the only one that allows us to be sure that the generated query is correct, (almost) avoiding runtime errors (to understand why "almost" see *Runtime errors*)

8.1.1 Conditions of the model

cql will only allow us to add conditions on the model we are querying, prohibiting the use of conditions from other models in the wrong place:

Listing 1: Correct

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
).Find()
```

Listing 2: Incorrect

```go
_, err := cql.Query[models.City](
    db,
    conditions.Country.Name.Is().Eq("Paris"),
).Find()
```

In this case, the compilation error will be:

```
cannot use conditions.Country.Name.Is().Eq("Paris")
(value of type condition.WhereCondition[models.Country]) as condition.Condition[models.
↪City]...
```

Similarly, conditions are checked when making joins:

Listing 3: Correct

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Country(
        conditions.Country.Name.Is().Eq("France"),
    ),
).Find()
```

Listing 4: Incorrect

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Country(
        conditions.City.Name.Is().Eq("France"),
    ),
).Find()
```

8.1.2 Name of an attribute or operator

Since the conditions are made using the auto-generated code, the attributes and methods used on it will only allow us to use attributes and operators that exist:

Listing 5: Correct

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
).Find()
```

Listing 6: Incorrect

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Namee.Is().Eq("Paris"),
).Find()
```

In this case, the compilation error will be:

```
conditions.City.Namee undefined (type conditions.cityConditions has no field or method␣
↪Namee)
```

8.1.3 Type of an attribute

cql not only verifies that the attribute used exists but also verifies that the value compared to the attribute is of the correct type:

Listing 7: Correct

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq("Paris"),
).Find()
```

Listing 8: Incorrect

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Name.Is().Eq(100),
).Find()
```

In this case, the compilation error will be:

```
cannot use 100 (untyped int constant) as string value in argument to conditions.City.
↪Name.Is().Eq
```

8.1.4 Type of an attribute (dynamic operator)

cql also checks that the type of the attributes is correct when using dynamic operators. In this case, the type of the two attributes being compared must be the same:

Listing 9: Correct

```go
_, err := cql.Query[models.City](
    db,
    conditions.City.Country(
        conditions.Country.Name.IsDynamic().Eq(conditions.City.Name.Value()),
    ),
).Find()
```

Listing 10: Incorrect

```
_, err := cql.Query[models.City](
    db,
    conditions.City.Country(
        conditions.Country.Name.IsDynamic().Eq(conditions.City.Population.Value()),
    ),
).Find()
```

In this case, the compilation error will be:

```
cannot use conditions.City.Population (variable of type condition.UpdatableField[models.
↪City, int]) as condition.FieldOfType[string] value in argument to conditions.Country.
↪Name.IsDynamic().Eq...
```

8.2 Runtime errors

Although all the above checks are at compile-time, there are still some possible cases that generate the following run-time errors:

- cql.ErrFieldModelNotConcerned **(1)**: generated when trying to use a model that is not related to the rest of the query (not joined).
- cql.ErrAppearanceMustBeSelected **(1)**: generated when you try to use a model that appears (is joined) more than once in the query without selecting which one you want to use (see *Appearance*).
- cql.ErrAppearanceOutOfRange **(1)**: generated when you try select an appearance number (with the Appearance method) greater than the number of appearances of a model. (see *Appearance*).
- cql.ErrFieldIsRepeated **(1)**: generated when a field is repeated inside a Set call (see *Update*).
- cql.ErrOnlyPreloadsAllowed: generated when trying to use conditions within a preload of collections (see *Collections*).
- cql.ErrUnsupportedByDatabase: generated when an attempt is made to use a method or function that is not supported by the database engine used.
- cql.ErrOrderByMustBeCalled: generated when in MySQL you try to do a delete/update with Limit but without using OrderBy.

Note: **(1)** errors avoided with *cqllint*.

However, these errors are discovered by CQL before the query is executed. In addition, CQL will add to the error clear information about the problem so that it is easy to fix, for example:

Listing 11: Query

```
_, err := cql.Query[models.Product](
    ts.db,
    conditions.Product.Int.Is().Eq(1),
).Descending(conditions.Seller.ID).Find()

fmt.Println(err)
```

Listing 12: Result

```
field's model is not concerned by the query (not joined); not concerned model: models.
↪Seller; method: Descending
```

CHAPTER

NINE

CQL-GEN

cql-gen is the command line tool that generates the conditions to query your objects. For each cql Model found in the input packages, a file containing all possible Conditions on that object will be generated, allowing you to use cql.

9.1 Installation

For simply installing it, use:

```
go install github.com/FrancoLiberali/cql/cql-gen@latest
```

Warning: The version of cql-gen used must be the same as the version of cql. You can install a specific version using *go install github.com/FrancoLiberali/cql/cql-gen@vX.Y.Z*, where X.Y.Z is the version number.

9.2 Conditions generation

While conditions can be generated executing cql-gen, it's recommended to use *go generate*:

Once cql-gen is installed, inside our project we will have to create a package called conditions (or another name if you wish) and inside it a file with the following content:

```
package conditions

//go:generate cql-gen ../models_path_1 ../models_path_2
```

where ../models_path_1 ../models_path_2 are the relative paths between the package conditions and the packages containing the definition of your models (can be only one).

Example file.

Now, from the root of your project you can execute:

```
go generate ./...
```

and the conditions for each of your models will be created in the conditions package.

9.3 Use of the conditions

After performing the conditions generation, your conditions package will have a replica of your models package, i.e. if, for example, the type models.MyModel is part of your models, the variable conditions.MyModel will be in the conditions package. This variable is called the condition model and it has:

- An attribute for each attribute of your original model with the same name (if models.MyModel.Name exists, then conditions.MyModel.Name is generated), that allows to use that attribute in your queries.
- A method for each relation of your original model with the same name (if models.MyModel.MyOtherModel exists, then conditions.MyModel.MyOtherModel() is generated), which will allow you to perform joins in your queries.
- Methods for *Preloading*.

Then, combining these conditions, the Connection Conditions (cql.And, cql.Or, cql.Not) you will be able to make all the queries you need in a safe way.

For details about querying, see *Query*.

CHAPTER

TEN

CQLLINT

cqllint is a Go linter that checks that cql queries will not generate run-time errors.

While, in most cases, queries created using cql are checked at compile time, there are still some cases that can generate run-time errors (see *Runtime errors*).

cqllint analyses the Go code written to detect these cases and fix them without the need to execute the query. It also adds other detections that would not generate runtime errors but are possible misuses of cql.

Note: At the moment, only the errors cql.ErrFieldModelNotConcerned, cql.ErrFieldIsRepeated, cql.ErrAppearanceMustBeSelected and cql.ErrAppearanceOutOfRange are detected.

We recommend integrating cqllint into your CI so that the use of cql ensures 100% that your queries will be executed correctly.

10.1 Installation

For simply installing it, use:

```
go install github.com/FrancoLiberali/cql/cqllint@latest
```

Warning: The version of cqllint used must be the same as the version of cql. You can install a specific version using *go install github.com/FrancoLiberali/cql/cqllint@vX.Y.Z*, where X.Y.Z is the version number.

10.2 Execution

cqllint can be used independently by running:

```
cqllint ./...
```

or using *go vet*:

```
go vet -vettool=$(which cqllint) ./...
```

10.3 Errors

10.3.1 ErrFieldModelNotConcerned

The simplest example this error case is trying to make a comparison with an attribute of a model that is not joined by the query:

Listing 1: example.go

```go
_, err := cql.Query[models.Brand](
    db,
    conditions.Brand.Name.IsDynamic().Eq(conditions.City.Name.Value()),
).Find()
```

If we execute this query we will obtain an error of type *cql.ErrFieldModelNotConcerned* with the following message:

```
field's model is not concerned by the query (not joined); not concerned model: models.
↪City; operator: Eq; model: models.Brand, field: Name
```

Now, if we run cqllint we will see the following report:

```
$ cqllint ./...
example.go:3: models.City is not joined by the query
```

10.3.2 ErrFieldIsRepeated

The simplest example this error case is trying to set the value of an attribute twice:

Listing 2: example.go

```go
_, err := cql.Update[models.Brand](
    db,
    conditions.Brand.Name.Is().Eq("nike"),
).Set(
    conditions.Brand.Name.Set().Eq("adidas"),
    conditions.Brand.Name.Set().Eq("puma"),
)
```

If we execute this query we will obtain an error of type *cql.ErrFieldIsRepeated* with the following message:

```
field is repeated; field: models.Brand.Name; method: Set
```

Now, if we run cqllint we will see the following report:

```
$ cqllint ./...
example.go:5: conditions.Brand.Name is repeated
example.go:6: conditions.Brand.Name is repeated
```

10.3.3 ErrAppearanceMustBeSelected

To generate this error we must join the same model more than once and not select the appearance number:

Listing 3: example.go

```go
1  _, err := cql.Query[models.Child](
2      db,
3      conditions.Child.Parent1(
4          conditions.Parent1.ParentParent(),
5      ),
6      conditions.Child.Parent2(
7          conditions.Parent2.ParentParent(),
8      ),
9      conditions.Child.ID.IsDynamic().Eq(conditions.ParentParent.ID.Value()),
10 ).Find()
```

If we execute this query we will obtain an error of type *cql.ErrAppearanceMustBeSelected* with the following message:

```
field's model appears more than once, select which one you want to use with Appearance; model: models.ParentParent; operator: Eq; model: models.Child, field: ID
```

Now, if we run cqllint we will see the following report:

```
$ cqllint ./...
example.go:9: models.ParentParent appears more than once, select which one you want to use with Appearance
```

10.3.4 ErrAppearanceOutOfRange

To generate this error we must use the Appearance method with a value greater than the number of appearances of a model:

Listing 4: example.go

```go
1  _, err := cql.Query[models.Phone](
2      db,
3      conditions.Phone.Brand(
4          conditions.Brand.Name.IsDynamic().Eq(conditions.Phone.Name.Appearance(1).Value()),
5      ),
6  ).Find()
```

If we execute this query we will obtain an error of type *cql.ErrAppearanceOutOfRange* with the following message:

```
selected appearance is bigger than field's model number of appearances; model: models.Phone; operator: Eq; model: models.Brand, field: Name
```

Now, if we run cqllint we will see the following report:

```
$ cqllint ./...
example.go:4: selected appearance is bigger than models.Phone's number of appearances
```

10.4 Misuses

Although some cases would not generate runtime errors, cqllint will detect them as they are possible misuses of cql.

10.4.1 Set the same value

This case occurs when making a Set of exactly the same value:

Listing 5: example.go

```
1 _, err := cql.Update[models.Brand](
2     db,
3     conditions.Brand.Name.Is().Eq("nike"),
4 ).Set(
5     conditions.Brand.Name.Set().Dynamic(conditions.Brand.Name.Value()),
6 )
```

If we run cqllint we will see the following report:

```
$ cqllint ./...
example.go:5: conditions.Brand.Name is set to itself
```

10.4.2 Unnecessary Appearance selection

This is the case when the Appearance method is used without being necessary, i.e. when the model appears only once:

Listing 6: example.go

```
1 _, err := cql.Query[models.Phone](
2     db,
3     conditions.Phone.Brand(
4         conditions.Brand.Name.IsDynamic().Eq(conditions.Phone.Name.Appearance(0).Value()),
5     ),
6 ).Find()
```

If we run cqllint we will see the following report:

```
$ cqllint ./...
example.go:4: Appearance call not necessary, models.Phone appears only once
```

CHAPTER

ELEVEN

QUERY

Read (query) operations are provided by cql via its compiled query system.

11.1 Query creation

To create a query you must use the cql.Query[models.MyModel] method, where models.MyModel is the model you expect this query to answer. This function takes as parameters the *transaction* on which to execute the query and the *Conditions*.

11.2 Transactions

To execute transactions, cql provides the function cql.Transaction. The function passed by parameter will be executed inside a gorm transaction (for more information visit https://gorm.io/docs/transactions.html). Using this method will also allow the transaction execution time to be logged.

11.3 Query methods

The object obtained using *cql.Query* has different methods that will allow you to obtain the results of the query:

11.3.1 Modifier methods

Modifier methods are those that modify the query in a certain way, affecting the results obtained: - Limit: specifies the number of models to be retrieved. - Offset: specifies the number of models to skip before starting to return the results. - Ascending: specifies an ascending order when retrieving models. - Descending: specifies a descending order when retrieving models from database.

11.3.2 Finishing methods

Finishing methods are those that cause the query to be executed and the result(s) of the query to be returned:

- Count: returns the amount of models that fulfill the conditions.
- First: finds the first model ordered by primary key.
- Take: finds the first model returned by the database in no specified order.
- Last: finds the last model ordered by primary key.
- FindOne: finds the only one model that matches given conditions or returns error if 0 or more than 1 are found.
- Find: finds list of models that meet the conditions.

11.4 Conditions

The set of conditions that are received by the *cql.Query* method form the cql compiled query system. It is so named because the conditions will verify at compile time that the query to be executed is correct.

These conditions are objects of type Condition that contain the necessary information to perform the queries in a safe way. They are generated from the definition of your models using *cql-gen*.

11.4.1 Examples

Filter by an attribute

In this example we query all MyModel that has "a_string" in the Name attribute.

```go
type MyModel struct {
    model.UUIDModel

    Name string
}

myModels, err := cql.Query[MyModel](
    gormDB,
    conditions.MyModel.Name.Is().Eq("a_string"),
).Find()
```

Filter by an attribute of a related model

In this example we query all MyModels whose related MyOtherModel has "a_string" in its Name attribute.

```go
type MyOtherModel struct {
    model.UUIDModel

    Name string
}

type MyModel struct {
    model.UUIDModel

    Related   MyOtherModel
```

(continues on next page)

(continued from previous page)

```
    RelatedID model.UUID
}

myModels, err := cql.Query[MyModel](
    gormDB,
    conditions.MyModel.Related(
        conditions.MyOtherModel.Name.Is().Eq("a_string"),
    ),
).Find()
```

Multiple conditions

In this example we query all MyModels that has a 4 in the Code attribute and whose related MyOtherModel has “a_string” in its Name attribute.

```
type MyOtherModel struct {
    model.UUIDModel

    Name string
}

type MyModel struct {
    model.UUIDModel

    Code int

    Related   MyOtherModel
    RelatedID model.UUID
}

myModels, err := cql.Query[MyModel](
    gormDB,
    conditions.MyModel.Code.Is().Eq(4),
    conditions.MyModel.Related(
        conditions.MyOtherModel.Name.Is().Eq("a_string"),
    ),
).Find()
```

11.5 Operators

The different operators to use inside your queries are defined by the methods of the FieldIs type, which is returned when calling the Is() method. Below you will find the complete list of available operators:

- Eq(value): Equal to
- NotEq(value): Not equal to
- Lt(value): Less than
- LtOrEq(value): Less than or equal to
- Gt(value): Greater than
- GtOrEq(value): Greater than or equal to

- Null()
- NotNull()
- Between(v1, v2): Equivalent to v1 < attribute < v2
- NotBetween(v1, v2): Equivalent to NOT (v1 < attribute < v2)
- Distinct(value)
- NotDistinct(value)
- In(values)
- NotIn(values)

For boolean attributes:

- True()
- NotTrue()
- False()
- NotFalse()
- Unknown(): unknown is null for booleans
- NotUnknown(): unknown is null for booleans

For string attributes:

- Like(pattern)

In addition to these, cql gives the possibility to use operators that are only supported by a certain database (outside the standard). For doing it, you must use the Custom method and give the operator as argument, for example:

```
conditions.MyModel.Code.Is().Custom(psql.ILike("_a%")),
```

These operators can be found in <https://pkg.go.dev/github.com/FrancoLiberali/cql/mysql>, <https://pkg.go.dev/github.com/FrancoLiberali/cql/sqlserver>, <https://pkg.go.dev/github.com/FrancoLiberali/cql/psql> and <https://pkg.go.dev/github.com/FrancoLiberali/cql/sqlite>.

You can also define your own operators following the condition.Operator interface.

CHAPTER

TWELVE

ADVANCED QUERY

12.1 Collections

cql also allows you to set conditions on a collection of models (one to many or many to many relationships):

Listing 1: Example model

```go
type Seller struct {
    model.UUIDModel

    Name string

    Company   *Company
    CompanyID *model.UUID // Company HasMany Seller (Company 0..1 -> 0..* Seller)
}

type Company struct {
    model.UUIDModel

    Sellers *[]Seller // Company HasMany Seller (Company 0..1 -> 0..* Seller)
}
```

Listing 2: Query

```
companies, err := cql.Query[Company](
    conditions.Company.Sellers.Any(
        conditions.Seller.Name.Is().Eq("franco"),
    ),
).Find()
```

The methods for collections are:

- None: generates a condition that is true if no model in the collection fulfills the conditions.
- Any: generates a condition that is true if at least one model in the collection fulfills the conditions.
- All: generates a condition that is true if all models in the collection fulfill the conditions (or is empty).

12.2 Dynamic operators

In *Query* we have seen how to use the operators to make comparisons between the attributes of a model and static values such as a string, a number, etc. But if we want to make comparisons between two or more attributes of the same type we need to use the dynamic operators. These, instead of a dynamic value, receive a Field, that is, an object that identifies the attribute with which the operation is to be performed.

These identifiers are also generated during the generation of conditions as attributes of the condition model (if models.MyModel.Name exists, then conditions.MyModel.Name is generated).

For example we query all MyModels that has the same value in its Name attribute that its related MyOtherModel's Name attribute.

Listing 3: Example model

```
type MyOtherModel struct {
    model.UUIDModel

    Name string
}

type MyModel struct {
    model.UUIDModel

    Name string

    Related   MyOtherModel
    RelatedID model.UUID
}
```

Listing 4: Query

```
myModels, err := cql.Query[MyModel](
    gormDB,
    conditions.MyModel.Related(
        conditions.MyOtherModel.Name.IsDynamic().Eq(conditions.MyModel.Name.Value()),
    ),
).Find()
```

Attention, when using dynamic operators the verification that the Field is concerned by the query is performed at run time, returning an error otherwise. For example:

Listing 5: Example model

```go
type MyOtherModel struct {
    model.UUIDModel

    Name string
}

type MyModel struct {
    model.UUIDModel

    Name string

    Related   MyOtherModel
    RelatedID model.UUID
}
```

Listing 6: Query

```go
myModels, err := cql.Query[MyModel](
    gormDB,
    conditions.MyModel.Name.IsDynamic().Eq(conditions.MyOtherModel.Name.Value()),
).Find()
```

will respond cql.ErrFieldModelNotConcerned in err.

All operators supported by cql that receive any value are available in their dynamic version after using the Dynamic() method of the FieldIs object.

12.2.1 Functions

When using dynamic operators it is also possible to apply functions on the values to be used. For example, if we seek to obtain the cities whose population represents at least half of the population of their country:

Listing 7: Example model

```go
type Country struct {
    model.UUIDModel

    Population int
}

type City struct {
    model.UUIDModel

    Population int

    Country   Country
    CountryID model.UUID
}
```

Listing 8: Query

```go
cities, err := cql.Query[City](
    gormDB,
    conditions.City.Country(
        conditions.Country.Population.IsDynamic().Lt(
            conditions.City.Population.Value().Times(2),
        ),
    ),
).Find()
```

12.3 Appearance

In case the attribute to be used is present more than once in the query, it will be necessary to select select its appearance number, to avoid getting the error cql.ErrAppearanceMustBeSelected. To do this, you must use the Appearance method of the field, as in the following example:

Listing 9: Example model

```go
type ParentParent struct {
    model.UUIDModel
}

type Parent1 struct {
    model.UUIDModel

    ParentParent   ParentParent
    ParentParentID model.UUID
}

type Parent2 struct {
    model.UUIDModel

    ParentParent   ParentParent
    ParentParentID model.UUID
}

type Child struct {
    model.UUIDModel

    Parent1   Parent1
    Parent1ID model.UUID

    Parent2   Parent2
    Parent2ID model.UUID
}
```

Listing 10: Query

```go
models, err := cql.Query[Child](
    gormDB,
```

(continues on next page)

(continued from previous page)

```
    conditions.Child.Parent1(
        conditions.Parent1.ParentParent(),
    ),
    conditions.Child.Parent2(
        conditions.Parent2.ParentParent(),
    ),
    conditions.Child.Name.IsDynamic().Eq(
        conditions.ParentParent.Name.Appearance(0).Value(), // choose the first (0)␣
↪appearance (made by conditions.Child.Parent1())
    ),
).Find()
```

12.4 Unsafe operators

In case you want to avoid the type validations performed by the operators, unsafe operators should be used. Although their use is not recommended, this can be useful when the database used allows operations between different types or when attributes of different types map at the same time in the database (see <https://gorm.io/docs/data_types.html>).

If it is neither of these two cases, the use of an unsafe operator will result in an error in the execution of the query that depends on the database used.

All operators supported by cql that receive any value are available in their unsafe version after using the IsUnsafe() method of the Field object.

12.5 Unsafe conditions (raw SQL)

In case you need to use operators that are not supported by cql (please create an issue in our repository if you think we have forgotten any), you can always run raw SQL with unsafe.NewCondition, as in the following example:

```
myModels, err := cql.Query[MyModel](
    gormDB,
    unsafe.NewCondition[MyModel]("%s.name = NULL"),
).Find()
```

As you can see in the example, “%s” can be used in the raw SQL to be replaced by the table name of the model to which the condition belongs.

Of course, its use is not recommended because it can generate errors in the execution of the query that will depend on the database used.

CHAPTER

THIRTEEN

PRELOADING

When doing a join, conditions can be applied on joined models but, by default, only the information of the main model is returned as a result. To also get the joined models, it is necessary to use the Preload() method.

13.1 Examples

Preload a related model

In this example we query all MyModels and preload whose related MyOtherModel.

```go
type MyOtherModel struct {
    model.UUIDModel
}

type MyModel struct {
    model.UUIDModel

    Related   MyOtherModel
    RelatedID model.UUID
}

myModels, err := cql.Query[MyModel](
    gormDB,
    conditions.MyModel.Related().Preload(),
).Find()
```

Nested preloads

```go
type Parent struct {
    model.UUIDModel
}

type MyOtherModel struct {
    model.UUIDModel

    Parent   Parent
    ParentID model.UUID
}

type MyModel struct {
```

(continues on next page)

(continued from previous page)

```go
    model.UUIDModel

    Related   MyOtherModel
    RelatedID model.UUID
}

myModels, err := cql.Query[MyModel](
    gormDB,
    conditions.MyModel.Related(
        conditions.MyOtherModel.Parent().Preload(),
    ),
).Find()
```

As we can see, it is not necessary to add the preload to all joins, it is enough to do it in the deepest one, to recover, in this example, both Related and Parent.

13.2 Relation getters

At the moment, with the PreloadConditions, we can choose whether or not to preload a relation. The problem is that once we get the result of the query, we cannot determine if a null value corresponds to the fact that the relation is really null or that the preload was not performed, which means a big risk of making decisions in our business logic on incomplete information.

For this reason, cql provides the Relation getters. These are methods that will be added to your models to safely navigate a relation, responding *cql.ErrRelationNotLoaded* in case you try to navigate a relation that was not loaded from the database. They are created in a file called cql.go in your model package when *generating conditions*.

Here is an example of its use:

```go
type MyOtherModel struct {
    model.UUIDModel
}

type MyModel struct {
    model.UUIDModel

    Related   MyOtherModel
    RelatedID model.UUID
}

myModel, err := cql.Query[MyModel](
    conditions.MyModel.Related().Preload(),
).FindOne()

if err == nil {
    firstRelated, err := myModel.GetRelated()
    if err == nil {
        // you can safely apply your business logic
    } else {
        // err is cql.ErrRelationNotLoaded
    }
}
```

Unfortunately, these relation getters cannot be created in all cases but only in those in which:

- The relation is made with an object directly instead of a pointer (which is not recommended as described *here*).
- The relation is made with pointers and the foreign key (typically the ID) is in the same model.
- The relation is made with a pointer to a list.

13.3 Preload collections

Model collections can also be preloaded (relations has many or many to many):

Listing 1: Example model

```go
type Seller struct {
    model.UUIDModel

    Company   *Company
    CompanyID *model.UUID // Company HasMany Seller (Company 0..1 -> 0..* Seller)
}

type Company struct {
    model.UUIDModel

    Sellers *[]Seller // Company HasMany Seller (Company 0..1 -> 0..* Seller)
}
```

Listing 2: Query

```go
company, err := cql.Query[Company](
    conditions.Company.Sellers.Preload(),
).FindOne()

if err == nil {
    sellers, err := company.GetSellers()
    if err == nil {
        // you can safely apply your business logic
    } else {
        // err is cql.ErrRelationNotLoaded
    }
}
```

Nested preloads can also be applied to preload model relationships within the collection:

Listing 3: Example model

```go
type Office struct {
    model.UUIDModel

    Seller   *Seller
    SellerID *model.UUID `gorm:"not null"` // Seller HasOne Office (Seller 1 -> 1 Office)
}

type Seller struct {
```

(continues on next page)

(continued from previous page)

```
    model.UUIDModel

    Office   *Office // Seller HasOne Office (Seller 1 -> 1 Office)

    Company   *Company
    CompanyID *model.UUID // Company HasMany Seller (Company 0..1 -> 0..* Seller)
}

type Company struct {
    model.UUIDModel

    Sellers *[]Seller // Company HasMany Seller (Company 0..1 -> 0..* Seller)
}
```

Listing 4: Query

```
company, err := cql.Query[Company](
    conditions.Company.Sellers.Preload(
        conditions.Seller.Office().Preload()
    ),
).FindOne()

if err == nil {
    sellers, err := company.GetSellers()
    if err == nil {
        for _, seller := range sellers {
            office, err := seller.GetOffice()
            if err == nil {
                // you can safely apply your business logic
            } else {
                // err is cql.ErrRelationNotLoaded
            }
        }
    } else {
        // err is cql.ErrRelationNotLoaded
    }
}
```

CHAPTER

FOURTEEN

CREATE

Create operations are made using gorm directly. For more information consult gorm documentation

CHAPTER

FIFTEEN

UPDATE

While update operations can still be performed using gorm's Save method (see gorm documentation), this is useful only if the model(s) to be updated have already been loaded from the database.

On the contrary, cql's Update method allows the update of all the models that meet the conditions entered without the need to load the information (via the direct execution of an UPDATE statement).

15.1 Update methods

Update operations are divided into two parts: the Update method and the Set method. In the first one, we must define the conditions that will determine which models will be updated. Here, the whole system of compilable queries is valid (for details visit *Query*). In the second one, we define the updates to be performed.

The object obtained using *cql.Update* has different methods that will allow you to modify the query:

15.1.1 Modifier methods

Modifier methods are those that modify the query in a certain way, affecting the models updated: - Limit: specifies the number of models to be updated. - Ascending: specifies an ascending order when updating models. - Descending: specifies a descending order when updating models. - Returning: specifies that the updated models must be fetched from the database after being updated. (not supported by MySQL). Preload of related data is also possible (not supported by SQLite).

15.1.2 Finishing methods

Finishing methods are those that cause the query to be executed:

- Set: defines the updates to be performed

15.1.3 Example

```go
type MyModel struct {
    model.UUIDModel

    Name string
}

updatedCount, err := cql.Update[MyModel](
```

(continues on next page)

(continued from previous page)

```
    gormDB,
    conditions.MyModel.Name.Is().Eq("a_string"),
).Set(
    conditions.MyModel.Name.Set().Eq("a_string_2"),
)
```

As you can see, the syntax for the Set method is similar to the queries system with the difference that the Set method must be used instead of Is.

For attributes that allow null (nullable values, pointers, nullable relations) the .Set().Null() method will also be available.

15.1.4 Joins

It is also possible to perform joins in the first part of the update (Update method):

```
type MyOtherModel struct {
    model.UUIDModel

    Name string
}

type MyModel struct {
    model.UUIDModel

    Name string

    Related   *MyOtherModel
    RelatedID *model.UUID
}

updatedCount, err := cql.Update[MyModel](
    gormDB,
    conditions.MyModel.Related(
        conditions.MyOtherModel.Name.Is().Eq("a_string"),
    ),
).Set(
    conditions.MyModel.Name.Set().Eq("a_string_2"),
)
```

Here the only limitation is that in the Set part, only the values of the initial model can be updated (not of the joined models).

This limitation is imposed by the database engines, with the exception of MySQL, which allows multiple tables to be updated at the same time. To do this, you use the SetMultiple method:

```
updatedCount, err := cql.Update[MyModel](
    gormDB,
    conditions.MyModel.Related(
        conditions.MyOtherModel.Name.Is().Eq("a_string"),
    ),
).SetMultiple(
    conditions.MyModel.Name.Set().Eq("a_string_2"),
```

(continues on next page)

(continued from previous page)

```
    conditions.MyOtherModel.Name.Set().Eq("a_string_2"),
)
```

CHAPTER

SIXTEEN

DELETE

While delete operations can still be performed using gorm's Delete method (see gorm documentation), this is useful only if the model(s) to be delete have already been loaded from the database.

On the contrary, cql's Delete method allows the deletion of all the models that meet the conditions entered without the need to load the information (via the direct execution of a DELETE statement).

16.1 Delete methods

Delete operations are divided into two parts: the Delete method and the Exec method. In the first one, we must define the conditions that will determine which models will be deleted. Here, the whole system of compilable queries is valid (for details visit *Query*).

The object obtained using *cql.Delete* has different methods that will allow you to modify the query:

16.1.1 Modifier methods

Modifier methods are those that modify the query in a certain way, affecting the models delete: - Limit: specifies the number of models to be deleted. (only supported by MySQL) - Ascending: specifies an ascending order when deleted models. (only supported by MySQL) - Descending: specifies a descending order when deleted models. (only supported by MySQL) - Returning: specifies that the models models must be fetched from the database after being deleted (the old data is returned) (not supported by MySQL). Preload of related data is also possible (not supported by SQLite).

16.1.2 Finishing methods

Finishing methods are those that cause the query to be executed:

- Exec: executes the delete

16.1.3 Example

```
type MyModel struct {
    model.UUIDModel

    Name string
}

deletedCount, err := cql.Delete[MyModel](
```

(continues on next page)

(continued from previous page)

```
    gormDB,
    conditions.MyModel.Name.Is().Eq("a_string"),
).Exec()
```

16.1.4 Joins

It is also possible to perform joins in the first part of the delete (Delete method):

```
type MyOtherModel struct {
    model.UUIDModel

    Name string
}

type MyModel struct {
    model.UUIDModel

    Name string

    Related   *MyOtherModel
    RelatedID *model.UUID
}

deletedCount, err := cql.Delete[MyModel](
    gormDB,
    conditions.MyModel.Related(
        conditions.MyOtherModel.Name.Is().Eq("a_string"),
    ),
).Exec()
```

Here the only limitation is that only the the initial models will be deleted (not of the joined models).

CHAPTER

SEVENTEEN

LOGGER

When connecting to the database, i.e. when creating the *gorm.DB* object, it is possible to configure the type of logger to use, the logging level, among others. As explained in the *connection section*, this can be done by using the *cql.Open* method:

```
gormDB, err = cql.Open(
  dialector,
  &gorm.Config{
    Logger: logger.Default,
  },
)
```

Any logger that complies with *logger.Interface* can be configured.

17.1 Log levels

The log levels provided by cql are the same as those of gorm:

- *logger.Error*: To only view error messages in case they occur during the execution of a sql query.
- *logger.Warn*: The previous level plus warnings for execution of queries and transactions that take longer than a certain time (configurable with SlowQueryThreshold and SlowTransactionThreshold respectively, 200ms by default).
- *logger.Info*: The previous level plus information messages for each query and transaction executed.

17.2 Transactions

For the logs corresponding to transactions (slow transactions and transaction execution) to be performed, it is necessary to use the cql.Transaction method.

17.3 Default logger

cql provides a default logger that will print Slow SQL and happening errors.

You can create one with the default configuration using (take into account that logger is github.com/FrancoLiberali/cql/logger and gormLogger is gorm.io/gorm/logger):

```
logger.Default
```

or use *logger.New* to customize it:

```
logger.New(logger.Config{
  LogLevel:                  gormLogger.Warn,
  SlowQueryThreshold:        200 * time.Millisecond,
  SlowTransactionThreshold:  200 * time.Millisecond,
  IgnoreRecordNotFoundError: false,
  ParameterizedQueries:      false,
  Colorful:                  true,
})
```

The LogLevel is also configurable via the *ToLogMode* method.

Example

```
example.go:30 [10.392ms] [rows:1] INSERT INTO "products" ("id","created_at","updated_at",
↪"deleted_at","string","int","float","bool") VALUES ('4e6d837b-5641-45c9-a028-
↪e5251e1a18b1','2023-07-21 17:19:59.563','2023-07-21 17:19:59.563',NULL,'',1,0.000000,
↪false)
```

17.4 Zap logger

cql provides the possibility to use zap as logger. For this, there is a package called *gormzap*. The information displayed by the zap logger will be the same as if we were using the default logger but in a structured form, with the following information:

- level: ERROR, WARN or DEBUG
- message:
  - query_error for errors during the execution of a query (ERROR)
  - query_slow for slow queries (WARN)
  - transaction_slow for slow transactions (WARN)
  - query_exec for query execution (DEBUG)
  - transaction_exec for transaction execution (DEBUG)
- error: <error_message> (for errors only)
- elapsed_time: query or transaction execution time
- rows_affected: number of rows affected by the query
- sql: query executed

You can create one with the default configuration using:

```
gormzap.NewDefault(zapLogger)
```

where *zapLogger* is a zap logger, or use *gormzap.New* to customize it:

```
gormzap.New(zapLogger, logger.Config{
  LogLevel:                  logger.Warn,
  SlowQueryThreshold:        200 * time.Millisecond,
  SlowTransactionThreshold:  200 * time.Millisecond,
  IgnoreRecordNotFoundError: false,
  ParameterizedQueries:      false,
})
```

The LogLevel is also configurable via the *ToLogMode* method. Any configuration of the zap logger is done directly during its creation following the zap documentation. Note that the zap logger has its own level setting, so the lower of the two settings will be the one finally used.

Example

```
DEBUG example.go:107  query_exec      {"elapsed_time": "3.291981ms", "rows_affected": "1
↪", "sql": "SELECT products.* FROM \"products\" WHERE products.int = 1 AND \"products\".
↪\"deleted_at\" IS NULL"}
```

CHAPTER EIGHTEEN

CONTRIBUTING

Thank you for your interest in CQL! This document provides the guidelines for how to contribute to the project through issues and pull-requests. Contributions can also come in additional ways such as commenting on issues or pull requests and more.

18.1 Issues

18.1.1 Issue types

There are 2 types of issues:

- Bug report: You've found a bug with the code, and want to report it, or create an issue to track the bug.
- Feature request: Used for items that propose a new idea or functionality. This allows feedback from others before code is written.

18.1.2 Before submitting

Before you submit an issue, make sure you've checked the following:

1. Check for existing issues
    - Before you create a new issue, please do a search in open issues to see if the issue or feature request has already been filed.
    - If you find your issue already exists, make relevant comments and add your reaction.
2. For bugs
    - It's not an environment issue.
    - You have as much data as possible. This usually comes in the form of logs and/or stacktrace.
3. You are assigned to the issue, a branch is created from the issue and the `wip` tag is added if you are also planning to develop the solution.

18.2 Pull Requests

All contributions come through pull requests. To submit a proposed change, follow this workflow:

1. Make sure there's an issue (bug report or feature request) opened, which sets the expectations for the contribution you are about to make
2. Assign yourself to the issue and add the `wip` tag
3. Fork the repo and create a new branch respecting the *naming policy* from the issue
4. Install the necessary *development environment*
5. Create your change and the corresponding *tests*
6. Update relevant documentation for the change in `docs/`
7. If changes are necessary in cql quickstart and cql tutorial, follow the same workflow there
8. Open a PR (and add links to the other repos' PR if they exist)
9. Wait for the CI process to finish and make sure all checks are green
10. A maintainer of the project will be assigned

18.2.1 Use work-in-progress PRs for early feedback

A good way to communicate before investing too much time is to create a "Work-in-progress" PR and share it with your reviewers. The standard way of doing this is to add a "[WIP]" prefix in your PR's title and assign the do-not-merge label. This will let people looking at your PR know that it is not well baked yet.

18.2.2 Branch naming policy

`[BRANCH_TYPE]/[BRANCH_NAME]`

- `BRANCH_TYPE` is a prefix to describe the purpose of the branch. Accepted prefixes are:
  - `feature`, used for feature development
  - `bugfix`, used for bug fix
  - `improvement`, used for refactor
  - `library`, used for updating library
  - `prerelease`, used for preparing the branch for the release
  - `release`, used for releasing project
  - `hotfix`, used for applying a hotfix on main
  - `poc`, used for proof of concept
- `BRANCH_NAME` is managed by this regex: `[a-z0-9._-]` (`_` is used as space character).

18.3 Code of Conduct

This project has adopted the Contributor Covenant Code of Conduct

CHAPTER

NINETEEN

DEVELOPING

This document provides the information you need to know before developing code for a pull request.

19.1 Environment

- Install go >= v1.20
- Install project dependencies: `go get`
- Install docker and compose plugin

19.2 Directory structure

This is the directory structure we use for the project:

- `docker/` : Contains the docker, docker-compose and configuration files for different environments.
- `docs/`: Contains the documentation showed for readthedocs.io.
- `test/`: Contains all the tests.

At the root of the project, you will find:

- The README.
- The changelog.
- The LICENSE file.

19.3 Tests

19.3.1 Dependencies

Running tests have some dependencies as: `gotestsum`, etc.. Install them with `make install_dependencies`.

19.3.2 Linting

We use `golangci-lint` for linting our code. You can test it with `make lint`. The configuration file is in the default path (`.golangci.yml`). The file `.vscode.settings.json.template` is a template for your `.vscode/settings.json` that formats the code according to our configuration.

19.3.3 Tests

We use the standard test suite in combination with github.com/stretchr/testify to do our testing. Tests have a database. CQL is tested on multiple databases. By default, the database used will be postgresql:

```
make test
```

To run the tests on another database you can use: `make test_postgresql`, `make test_cockroachdb`, `make test_mysql`, `make test_sqlite`, `make test_sqlserver`. All of them will be verified by our continuous integration system.

19.4 Requirements

To be acceptable, contributions must:

- Have a good quality of code, based on https://go.dev/doc/effective_go.
- Have at least 80 percent new code coverage (although a higher percentage may be required depending on the importance of the feature). The tests that contribute to coverage are unit tests and integration tests.
- The features defined in the PR base issue must be explicitly tested by tests.

19.5 Use of Third-party code

Third-party code must include licenses.

CHAPTER

TWENTY

MAINTAINING

This document is intended for CQL maintainers only.

20.1 How to release

Release tag are only done on the `main` branch. We use Semantic Versioning as guideline for the version management.

Steps to release:

- Create a new branch labeled `release/vX.Y.Z` from the latest `main`.
- Improve the version number in `cql-gen/version/version.go` and `cqllint/version/version.go`.
- Commit the modifications with the label `Release version X.Y.Z`.
- Create a pull request on github for this branch into `main`.
- Once the pull request validated and merged, tag the `main` branch using `./create_tag.sh X.Y.Z`.