GraphQL Compiler

Jun 02, 2021

Contents

1 Getting Started **3**
1.1 Generating the necessary schema info . . . 3
1.2 Query Compilation and Execution . . . 4

2 Features **5**
2.1 Language Specification . . . 5
2.2 Supported Databases . . . 32
2.3 Advanced Features . . . 37
2.4 About GraphQL compiler . . . 44

GraphQL compiler is a library that simplifies data querying and exploration by exposing one simple query language to target multiple database backends. The query language is:

Written in valid GraphQL syntax Since it uses GraphQL syntax, the user get access to the entire GraphQL ecosystem, including the typeahead capabilities and query validation capabilities of GraphiQL, user friendly error messages from the reference GraphQL python implementation, and more.

Directly compiled to the target database language By compiling instead of interpreting the query language, the compiler highly improves query performance and empowers the user with the ability to write deep and complex queries. Furthermore, by using schema information from the target database, the compiler is able to extensively validate queries, often more so than the DB-API, (e.g. `pymssql`).

CHAPTER 1

Getting Started

1.1 Generating the necessary schema info

To use GraphQL compiler the first thing one needs to do is to generate the schema info from the underlying database as in the example below. Even though the example targets an OrientDB database, it is meant as a generic schema info generation example. See the homepage of your target database for more instructions on how to generate the necessary schema info.

```
from graphql_compiler import (
    get_graphql_schema_from_orientdb_schema_data
)
from graphql_compiler.schema_generation.orientdb.utils import ORIENTDB_SCHEMA_RECORDS_
↪QUERY

client = your_function_that_returns_a_pyorient_client()
schema_records = client.command(ORIENTDB_SCHEMA_RECORDS_QUERY)
schema_data = [record.oRecordData for record in schema_records]
schema, type_equivalence_hints = get_graphql_schema_from_orientdb_schema_data(schema_
↪data)
```

In the snippet above the are two pieces of schema info:

- `schema` which represents the database using GraphQL's type system.
- `type_equivalence_hints` which helps deal with GraphQL's lack of concrete inheritance, (see *schema types* for more info).

When compiling, these will need to be bundled in a `CommonSchemaInfo` object.

Besides representing the database schema, a GraphQL schema includes other metadata such as a list of custom scalar types used by the compiler. We'll talk more about this metadata in *schema types*. For now let's focus on how a database schema might be represented in a GraphQL schema:

```
type Animal {
    name: String
```

(continues on next page)

(continued from previous page)

```
    out_Animal_LivesIn: [Continent]
}

type Continent {
    name: String
    in_AnimalLivesIn: [Animal]
}
```

In the GraphQL schema above:

- `Animal` represents a concrete, (non-abstract), vertex type. For relational databases, we think of tables as the concrete vertex types.
- `name` is a **property field** which represents a property of the `Animal` vertex type. Think of property fields as leaf fields that represent concrete data.
- `out_Animal_LivesIn` is a **vertex field** which represents an outbound edge to a vertex type in the graph. For graph databases, edges can be automatically generated from the database schema. However, for relational databases, edges currently have to be manually specified. See *SQL* for more information.

1.2 Query Compilation and Execution

Once we have the schema info we can write the following query to get the names of all the animals that live in Africa:

```
graphql_query = """
{
    Animal {
        name @output(out_name: "animal_name")
        out_Animal_LivesIn {
            name @filter(op_name: "=", value: ["$continent"])
        }
    }
}
"""
parameters = {'continent': 'Africa'}
```

There are a couple of things to notice about queries:

- All queries start with a vertex type, (e.g. `Animal`), and expand to other vertex types using vertex fields.
- **Directives** specify the semantics of a query. `@output` indicates the properties whose values should be returned. `@filter` specifies a filter operation.

Finally, with the GraphQL query and its parameters at hand, we can use the compiler to obtain a query that we can directly execute against OrientDB.

```
from graphql_compiler import graphql_to_match

compilation_result = graphql_to_match(
    schema, graphql_query, parameters, type_equivalence_hints)

# Executing query assuming a pyorient client. Other clients may have a different␣
↪interface.
print([result.oRecordData for result in client.query(query)])
# [{'animal_name': 'Elephant'}, {'animal_name': 'Lion'}, ...]
```

CHAPTER 2

Features

2.1 Language Specification

To learn more about the language specification see:

- *Definitions*, for the definitions of key terms that we use to define the language.
- *Schema Types*, for information about the full breadth of schema types that we use to represent database schemas and how to interact with them using GraphQL queries.
- *Query Directives*, to learn more about the available directives and how to use them to create powerful queries.

2.1.1 Definitions

- **Vertex field**: A field corresponding to a vertex in the graph. In the below example, `Animal` and `out_Entity_Related` are vertex fields. The `Animal` field is the field at which querying starts, and is therefore the **root vertex field**. In any scope, fields with the prefix `out_` denote vertex fields connected by an outbound edge, whereas ones with the prefix `in_` denote vertex fields connected by an inbound edge.

```
{
    Animal {
        name @output(out_name: "name")
        out_Entity_Related {
            ... on Species {
                description @output(out_name: "description")
            }
        }
    }
}
```

- **Property field**: A field corresponding to a property of a vertex in the graph. In the above example, the `name` and `description` fields are property fields. In any given scope, **property fields must appear before vertex fields**.

- **Result set**: An assignment of vertices in the graph to scopes (locations) in the query. As the database processes the query, new result sets may be created (e.g. when traversing edges), and result sets may be discarded when they do not satisfy filters or type coercions. After all parts of the query are processed by the database, all remaining result sets are used to form the query result, by taking their values at all properties marked for output.
- **Scope**: The part of a query between any pair of curly braces. The compiler infers the type of each scope. For example, in the above query, the scope beginning with `Animal {` is of type `Animal`, the one beginning with `out_Entity_Related {` is of type `Entity`, and the one beginning with `... on Species {` is of type `Species`.
- **Type coercion**: An operation that produces a new scope of narrower type than the scope in which it exists. Any result sets that cannot satisfy the narrower type are filtered out and not returned. In the above query, `... on Species` is a type coercion which takes its enclosing scope of type `Entity`, and coerces it into a narrower scope of type `Species`. This is possible since `Entity` is an interface, and `Species` is a type that implements the `Entity` interface.

2.1.2 Schema Types

A GraphQL schema might look like the one below. Do not be intimidated by the number of components since we will immediately proceed to dissect the schema.

```
schema {
    query: RootSchemaQuery
}

type RootSchemaQuery {
    Animal: [Animal]
    Entity: [Entity]
    Food: [Food]
    Species: [Species]
    Toy: [Toy]
}

directive @filter(op_name: String!, value: [String!]) on FIELD | INLINE_FRAGMENT

directive @tag(tag_name: String!) on FIELD

directive @output(out_name: String!) on FIELD

directive @output_source on FIELD

directive @optional on FIELD

directive @recurse(depth: Int!) on FIELD

directive @fold on FIELD

scalar Date

scalar DateTime

scalar Decimal

type Animal implements Entity {
    _x_count: Int
    uuid: ID
```

(continues on next page)

(continued from previous page)

```
    name: String
    alias: [String]
    color: String
    birthday: Date
    net_worth: Decimal
    in_Animal_ParentOf: [Animal]
    out_Animal_ParentOf: [Animal]
    in_Entity_Related: [Entity]
    out_Entity_Related: [Entity]
    out_Animal_OfSpecies: [Species]
    out_Animal_PlaysWith: [Toy]
}

type Food implements Entity {
    _x_count: Int
    uuid: ID
    name: String
    alias: [String]
    in_Entity_Related: [Entity]
    out_Entity_Related: [Entity]
    in_Species_Eats: [Species]
}

type Species implements Entity {
    _x_count: Int
    uuid: ID
    name: String
    alias: [String]
    in_Animal_OfSpecies: [Animal]
    in_Entity_Related: [Entity]
    out_Entity_Related: [Entity]
    in_Species_Eats: [Species]
    out_Species_Eats: [Union__Food__Species]
}

type Toy {
    _x_count: Int
    uuid: ID
    name: String
    in_Animal_PlaysWith: [Animal]
}

interface Entity {
    _x_count: Int
    uuid: ID
    name: String
    alias: [String]
    in_Entity_Related: [Entity]
    out_Entity_Related: [Entity]
}

union Union__Food__Species = Food | Species
```

Note: A GraphQL schema can be serialized with the `print_schema` function from the `graphql.utils.schema_printer` module.

Objects types and fields

The core components of a GraphQL schema are GraphQL object types. They conceptually represent the concrete vertex types in the underlying database. For relational databases, we think of the tables as the concrete vertex types.

Lets go over a toy example of a GraphQL object type:

```
type Toy {
    _x_count: Int
    name: String
    in_Animal_PlaysWith: [Animal]
}
```

Here are some of the details:

- `_x_count`: is a *meta field*. Meta fields are an advanced compiler feature.
- `name` is a **property field** that represents concrete data.
- `in_Animal_PlaysWith` is a **vertex field** representing an inbound edge.
- `String` is a built-in GraphQL scalar type.
- `[Animal]` is a GraphQL list representing a list of `Animal` objects.

Directives

Directives are keywords that modify query execution. The compiler includes a list of directives, which we'll talk about more in the *query directives* section. For now lets see how they are defined by looking at an example:

```
directive @output(out_name: String!) on FIELD
```

- `@output` defines the directive name.
- `out_name: String!` is a GraphQL argument. The `!` indicates that it must not be null.
- `on FIELD` defines where the directive can be located. According to the definition, this directive can only be located next to fields. The compiler might have additional restrictions for where a query can be located.

Scalar types

The compiler uses the built-in GraphQL scalar types as well as three custom scalar types:

- `DateTime` represents timezone-naive second-accuracy timestamps.
- `Date` represents day-accuracy date objects.
- `Decimal` is an arbitrary-precision decimal number object useful for representing values that should never be rounded, such as currency amounts.

Operation types

GraphQL allows for three operation types *query*, *mutation* and *subscription*. The compiler only allows for read-only *query* operation types as shown in the code snippet below:

```
schema {
    query: RootSchemaQuery
}
```

A query may begin in any of the **root vertex types** specified by the special `RootSchemaQuery` object type:

```
type RootSchemaQuery {
    Animal: [Animal]
    Entity: [Entity]
    Food: [Food]
    Species: [Species]
    Toy: [Toy]
}
```

Inheritance

The compiler uses interface and union types in representing the inheritance structure of the underlying schema. Some database backends do not support inheritance, (e.g. SQL), so this feature is only supported for certain backends.

Interface types

Object types may declare that they *implement* an interface type, meaning that they contain all property and vertex fields that the interface declares. In many programming languages, this concept is called interface inheritance or abstract inheritance. The compiler uses interface implementation in the GraphQL schema to model the abstract inheritance in the underlying database.

```
interface Entity {
     _x_count: Int
     name: String
     alias: [String]
     in_Entity_Related: [Entity]
     out_Entity_Related: [Entity]
 }

 type Food implements Entity {
     _x_count: Int
     name: String
     alias: [String]
     in_Entity_Related: [Entity]
     out_Entity_Related: [Entity]
     in_Species_Eats: [Species]
 }
```

Querying an interface type without any type coercion returns all of the the objects implemented by the interface. For instance, the following query returns the name of all `Food`, `Species` and `Animal` objects.

```
{
   Entity {
      name @output(out_name: "entity_name")
   }
}
```

Union types and `type_equivalence_hints`

GraphQL's type system does not allow object types to inherit other object types (i.e. it has no notion of concrete inheritance). However, to model the database schema of certain backends and to emit the right query in certain cases, the compiler needs to have a notion of the underlying concrete inheritance.

In order to work around this limitation, the GraphQL compiler uses GraphQL union types as means of listing the subclasses of an object with multiple implicit subclasses. It also takes in a `type_equivalence_hints` parameter to match an object type with the union type listing its subclasses.

For example, suppose `Food` and `Species` are concrete types and `Food` is a superclass of `Species` in an OrientDB schema. Then the GraphQL schema info generation function would generate a union type in the schema

```
union Union__Food__Species = Food | Species
```

as well an entry in `type_equivalence_hints` mapping `Food` to `Union_Food_Species`.

To query an union type, one must always type coerce to one of the encompassed object types as illustrated in the section below.

Type coercions

Type coercions are operations than can be run against interfaces and unions to create a new scope whose type is different than the type of the enclosing scope of the coercion. Type coercions are represented with GraphQL inline fragments.

Example Use

```
{
    Species {
        name @output(out_name: "species_name")
        out_Species_Eats {
            ... on Food {
                name @output(out_name: "food_name")
            }
        }
    }
}
```

Here, the `out_Species_Eats` vertex field is of the `Union__Food__FoodOrSpecies__Species` union type. To proceed with the query, the user must choose which of the types in the `Union__Food__FoodOrSpecies__Species` union to use. In this example, `... on Food` indicates that the `Food` type was chosen, and any vertices at that scope that are not of type `Food` are filtered out and discarded.

```
{
    Species {
        name @output(out_name: "species_name")
        out_Entity_Related {
            ... on Species {
                name @output(out_name: "entity_name")
            }
        }
    }
}
```

In this query, the `out_Entity_Related` is of `Entity` type. However, the query only wants to return results where the related entity is a `Species`, which `... on Species` ensures is the case.

Constraints and Rules

- Must be the only selection in scope. No field may exist in the same scope as a type coercion. No scope may contain more than one type coercion.

Meta fields

Meta fields are fields that do not represent a property/column in the underlying vertex type. They are also an advanced compiler feature. Before continuing, readers should familiarize themselves with the various *query directives* supported by the compiler.

__typename

The compiler supports the standard GraphQL meta field `__typename`, which returns the runtime type of the scope where the field is found. Assuming the GraphQL schema matches the database's schema, the runtime type will always be a subtype of (or exactly equal to) the static type of the scope determined by the GraphQL type system. Below, we provide an example query in which the runtime type is a subtype of the static type, but is not equal to it.

The `__typename` field is treated as a property field of type `String`, and supports all directives that can be applied to any other property field.

Example Use

```
{
    Entity {
        __typename @output(out_name: "entity_type")
        name @output(out_name: "entity_name")
    }
}
```

This query returns one row for each `Entity` vertex. The scope in which `__typename` appears is of static type `Entity`. However, `Animal` is a type of `Entity`, as are `Species`, `Food`, and others. Vertices of all subtypes of `Entity` will therefore be returned, and the `entity_type` column that outputs the `__typename` field will show their runtime type: `Animal`, `Species`, `Food`, etc.

_x_count

The `_x_count` meta field is a non-standard meta field defined by the GraphQL compiler that makes it possible to interact with the *number* of elements in a scope marked `@fold`. By applying directives like `@output` and `@filter` to this meta field, queries can output the number of elements captured in the `@fold` and filter down results to select only those with the desired fold sizes.

We use the `_x_` prefix to signify that this is an extension meta field introduced by the compiler, and not part of the canonical set of GraphQL meta fields defined by the GraphQL specification. We do not use the GraphQL standard double-underscore (`__`) prefix for meta fields, since all names with that prefix are explicitly reserved and prohibited from being used in directives, fields, or any other artifacts.

Adding the `_x_count` meta field to your schema

Since the `_x_count` meta field is not currently part of the GraphQL standard, it has to be explicitly added to all interfaces and types in your schema. There are two ways to do this.

The preferred way to do this is to use the `EXTENDED_META_FIELD_DEFINITIONS` constant as a starting point for building your interfaces' and types' field descriptions:

```
from graphql import GraphQLInt, GraphQLField, GraphQLObjectType, GraphQLString
from graphql_compiler import EXTENDED_META_FIELD_DEFINITIONS
fields = EXTENDED_META_FIELD_DEFINITIONS.copy()
fields.update({
    'foo': GraphQLField(GraphQLString),
    'bar': GraphQLField(GraphQLInt),
    # etc.
})
graphql_type = GraphQLObjectType('MyType', fields)
# etc.
```

If you are not able to programmatically define the schema, and instead simply have a pre-made GraphQL schema object that you are able to mutate, the alternative approach is via the `insert_meta_fields_into_existing_schema()` helper function defined by the compiler:

```
# assuming that existing_schema is your GraphQL schema object
insert_meta_fields_into_existing_schema(existing_schema)
# existing_schema was mutated in-place and all custom meta-fields were added
```

Example Use

```
{
    Animal {
        name @output(out_name: "name")
        out_Animal_ParentOf @fold {
            _x_count @output(out_name: "number_of_children")
            name @output(out_name: "child_names")
        }
    }
}
```

This query returns one row for each `Animal` vertex. Each row contains its name, and the number and names of its children. While the output type of the `child_names` selection is a list of strings, the output type of the `number_of_children` selection is an integer.

```
{
    Animal {
        name @output(out_name: "name")
        out_Animal_ParentOf @fold {
            _x_count @filter(op_name: ">=", value: ["$min_children"])
                    @output(out_name: "number_of_children")
            name @filter(op_name: "has_substring", value: ["$substr"])
                 @output(out_name: "child_names")
        }
    }
}
```

Here, we've modified the above query to add two more filtering constraints to the returned rows:

- child Animal vertices must contain the value of $substr as a substring in their name, and
- Animal vertices must have at least $min_children children that satisfy the above filter.

Importantly, any filtering on _x_count is applied *after* any other filters and type coercions that are present in the @fold in question. This order of operations matters a lot: selecting Animal vertices with 3+ children, then filtering the children based on their names is not the same as filtering the children first, and then selecting Animal vertices that have 3+ children that matched the earlier filter.

Constraints and Rules

- The _x_count field is only allowed to appear within a vertex field marked @fold.
- Filtering on _x_count is always applied *after* any other filters and type coercions present in that @fold.
- Filtering or outputting the value of the _x_count field must always be done at the innermost scope of the @fold. It is invalid to expand vertex fields within a @fold after filtering or outputting the value of the _x_count meta field.

How is filtering on _x_count different from @filter with has_edge_degree?

The has_edge_degree filter allows filtering based on the number of edges of a particular type. There are situations in which filtering with has_edge_degree and filtering using = on _x_count produce equivalent queries. Here is one such pair of queries:

```
{
    Species {
        name @output(out_name: "name")
        in_Animal_OfSpecies @filter(op_name: "has_edge_degree", value: ["$num_animals
↪"]) {
            uuid
        }
    }
}
```

and

```
{
    Species {
        name @output(out_name: "name")
        in_Animal_OfSpecies @fold {
            _x_count @filter(op_name: "=", value: ["$num_animals"])
        }
    }
}
```

In both of these queries, we ask for the names of the Species vertices that have precisely $num_animals members. However, we have expressed this question in two different ways: once as a property of the Species vertex ("the degree of the in_Animal_OfSpecies is $num_animals"), and once as a property of the list of Animal vertices produced by the @fold ("the number of elements in the @fold is $num_animals").

When we add additional filtering within the Animal vertices of the in_Animal_OfSpecies vertex field, this distinction becomes very important. Compare the following two queries:

```
{
    Species {
        name @output(out_name: "name")
        in_Animal_OfSpecies @filter(op_name: "has_edge_degree", value: ["$num_animals
↪"]) {
            out_Animal_LivesIn {
                name @filter(op_name: "=", value: ["$location"])
            }
        }
    }
}
```

versus

```
{
    Species {
        name @output(out_name: "name")
        in_Animal_OfSpecies @fold {
            out_Animal_LivesIn {
                _x_count @filter(op_name: "=", value: ["$num_animals"])
                name @filter(op_name: "=", value: ["$location"])
            }
        }
    }
}
```

In the first, for the purposes of the `has_edge_degree` filtering, the location where the animals live is irrelevant: the `has_edge_degree` only makes sure that the `Species` vertex has the correct number of edges of type `in_Animal_OfSpecies`, and that's it. In contrast, the second query ensures that only `Species` vertices that have `$num_animals` animals that live in the selected location are returned – the location matters since the `@filter` on the `_x_count` field applies to the number of elements in the `@fold` scope.

2.1.3 Query Directives

@optional

Without this directive, when a query includes a vertex field, any results matching that query must be able to produce a value for that vertex field. Applied to a vertex field, this directive prevents result sets that are unable to produce a value for that field from being discarded, and allowed to continue processing the remainder of the query.

Example Use

```
{
    Animal {
        name @output(out_name: "name")
        out_Animal_ParentOf @optional {
            name @output(out_name: "child_name")
        }
    }
}
```

For each `Animal`:

- if it is a parent of another animal, at least one row containing the parent and child animal's names, in the `name` and `child_name` columns respectively;
- if it is not a parent of another animal, a row with its name in the `name` column, and a `null` value in the `child_name` column.

Constraints and Rules

- `@optional` can only be applied to vertex fields, except the root vertex field.
- It is allowed to expand vertex fields within an `@optional` scope. However, doing so is currently associated with a performance penalty in `MATCH`.
- `@recurse`, `@fold`, or `@output_source` may not be used at the same vertex field as `@optional`.
- `@output_source` and `@fold` may not be used anywhere within a scope marked `@optional`.

If a given result set is unable to produce a value for a vertex field marked `@optional`, any fields marked `@output` within that vertex field return the `null` value.

When filtering (via `@filter`) or type coercion (via e.g. `... on Animal`) are applied at or within a vertex field marked `@optional`, the `@optional` is given precedence:

- If a given result set cannot produce a value for the optional vertex field, it is preserved: the `@optional` directive is applied first, and no filtering or type coercion can happen.
- If a given result set is able to produce a value for the optional vertex field, the `@optional` does not apply, and that value is then checked against the filtering or type coercion. These subsequent operations may then cause the result set to be discarded if it does not match.

For example, suppose we have two `Person` vertices with names `Albert` and `Betty` such that there is a `Person_Knows` edge from `Albert` to `Betty`.

Then the following query:

```
{
  Person {
    out_Person_Knows @optional {
      name @filter(op_name: "=", value: ["$name"])
    }
    name @output(out_name: "person_name")
  }
}
```

with runtime parameter

```
{
  "name": "Charles"
}
```

would output

```
[
    { name: 'Betty' },
]
```

because the `Person_Knows` edge from `Albert` to `Betty` satisfies the `@optional` directive, but `Betty` doesn't match the filter checking for a node with name `Charles`.

However, if no such `Person_Knows` edge existed from `Albert`, then the output would be

```
[
    { name: 'Albert' },
    { name: 'Betty' },
]
```

because no such edge can satisfy the @optional directive, and no filtering happens. In both examples, Betty is always returned because Betty does not have any outgoing Person_Knows edges.

@output

Denotes that the value of a property field should be included in the output. Its out_name argument specifies the name of the column in which the output value should be returned.

Example Use

```
{
    Animal {
        name @output(out_name: "animal_name")
    }
}
```

This query returns the name of each Animal in the graph, in a column named animal_name.

Constraints and Rules

- @output can only be applied to property fields.
- The value provided for out_name may only consist of upper or lower case letters (A-Z, a-z), or underscores (_).
- The value provided for out_name cannot be prefixed with ___ (three underscores). This namespace is reserved for compiler internal use.
- For any given query, all out_name values must be unique. In other words, output columns must have unique names.

If the property field marked @output exists within a scope marked @optional, result sets that are unable to assign a value to the optional scope return the value null as the output of that property field.

@fold

Applying @fold on a scope "folds" all outputs from within that scope: rather than appearing on separate rows in the query result, the folded outputs are coalesced into parallel lists starting at the scope marked @fold.

It is also possible to output or apply filters to the number of results captured in a @fold. The _x_count meta field that is available within @fold scopes represents the number of elements in the fold, and may be filtered or output as usual. As _x_count represents a count of elements, marking it @output will produce an integer value. See the *_x_count* section for more details.

Example Use

```
{
    Animal {
        name @output(out_name: "animal_name")
        out_Entity_Related @fold {
            ... on Location {
                _x_count @output(out_name: "location_count")
                name @output(out_name: "location_names")
            }
        }
    }
}
```

Each returned row has three columns: `animal_name` with the name of each `Animal` in the graph, `location_count` with the related locations for that `Animal`, and `location_names` with a list of the names of all related locations of the `Animal` named `animal_name`. If a given `Animal` has no related locations, its `location_names` list is empty and the `location_count` value is 0.

Constraints and Rules

- `@fold` can only be applied to vertex fields, except the root vertex field.
- May not exist at the same vertex field as `@recurse`, `@optional`, or `@output_source`.
- Any scope that is either marked with `@fold` or is nested within a `@fold` marked scope, may expand at most one vertex field.
- "No no-op `@fold` scopes": within any `@fold` scope, there must either be at least one field that is marked `@output`, or there must be a `@filter` applied to the `_x_count` field.
- All `@output` fields within a `@fold` traversal must be present at the innermost scope. It is invalid to expand vertex fields within a `@fold` after encountering an `@output` directive.
- `@tag`, `@recurse`, `@optional`, `@output_source` and `@fold` may not be used anywhere within a scope marked `@fold`.
- The `_x_count` meta field may only appear at the innermost scope of a `@fold` marked scope.
- Marking the `_x_count` meta field with an `@output` produces an integer value corresponding to the number of results within that fold.
- Marking for `@output` any field other than the `_x_count` meta field produces a list of results, where the number of elements in that list is equal to the value of the `_x_count` meta field, if it were selected for output.
- If multiple fields (other than `_x_count`) are marked `@output`, the resulting output lists are parallel: the `i`th element of each such list is the value of the corresponding field of the `i`th element of the `@fold`, for some fixed order of elements in that `@fold`. The order of elements within the output of a `@fold` is only fixed for a particular execution of a given query, for the results of a given `@fold` that are part of a single result set. There is no guarantee of consistent ordering of elements for the same `@fold` in any of the following situations:
  - across two or more result sets that are both the result of the execution of the same query;
  - across different executions of the same query, or
  - across different queries that contain the same `@fold` scope.
- Use of type coercions or `@filter` at or within the vertex field marked `@fold` is allowed. The order of operations is conceptually as follows:

  - First, type coercions and filters (except `@filter` on the `_x_count` meta field) are applied, and any data that does not satisfy such coercions and filters is discarded. At this point, the size of the fold (i.e. its number of results) is fixed.
  - Then, any `@filter` directives on the `_x_count` meta field are applied, allowing filtering of result sets based on the fold size. Any result sets that do not match these filters are discarded.
  - Finally, if the result set was not discarded by the previous step, `@output` directives are processed, selecting folded data for output.
- If the compiler is able to prove that a type coercion in the `@fold` scope is actually a no-op, it may optimize it away.

Example

The following GraphQL is *not allowed* and will produce a `GraphQLCompilationError`. This query is *invalid* for two separate reasons:

- It expands vertex fields after an `@output` directive (outputting `animal_name`)
- The `in_Animal_ParentOf` scope, which is within a scope marked `@fold`, expands two vertex fields instead of at most one.

```
{
    Animal {
        out_Animal_ParentOf @fold {
            name @output(out_name: "animal_name")
            in_Animal_ParentOf {
                out_Animal_OfSpecies {
                    uuid @output(out_name: "species_id")
                }
                out_Entity_Related {
                    ... on Animal {
                        name @output(out_name: "relative_name")
                    }
                }
            }
        }
    }
}
```

The following GraphQL query is similarly *not allowed* and will produce a `GraphQLCompilationError`, since the `_x_count` field is not within the innermost scope in the `@fold`.

```
{
    Animal {
        out_Animal_ParentOf @fold {
            _x_count @output(out_name: "related_count")
            out_Entity_Related {
                ... on Animal {
                    name @output(out_name: "related_name")
                }
            }
        }
    }
}
```

Moving the `_x_count` field to the innermost scope results in the following valid use of `@fold`:

```
{
    Animal {
        out_Animal_ParentOf @fold {
            out_Entity_Related {
                ... on Animal {
                    _x_count @output(out_name: "related_count")
                    name @output(out_name: "related_name")
                }
            }
        }
    }
}
```

Here is an example of query whose `@fold` does not output any data; it returns the names of all animals that have more than `count` children whose names contain the substring `substr`:

```
{
    Animal {
        name @output(out_name: "animal_name")
        out_Animal_ParentOf @fold {
            _x_count @filter(op_name: ">=", value: ["$count"])
            name @filter(op_name: "has_substring", value: ["$substr"])
        }
    }
}
```

@tag

The `@tag` directive enables filtering based on values encountered elsewhere in the same query. Applied on a property field, it assigns a name to the value of that property field, allowing that value to then be used as part of a `@filter` directive.

To supply a tagged value to a `@filter` directive, place the tag name (prefixed with a `%` symbol) in the `@filter`'s `value` array. See *Passing parameters* for more details.

Example Use

```
{
    Animal {
        limbs @tag(tag_name: "parent_limbs")
        out_Animal_ParentOf {
            limbs @filter(op_name: "<", value: ["%parent_limbs"])
            name @output(out_name: "child_name")
        }
    }
}
```

Each result returned by this query contains the name of an `Animal` who is a child of another animal and has fewer limbs than its parent.

Constraints and Rules

- `@tag` can only be applied to property fields.

- The value provided for `tag_name` may only consist of upper or lower case letters (`A-Z`, `a-z`), or underscores (`_`).
- For any given query, all `tag_name` values must be unique.
- Cannot be applied to property fields within a scope marked `@fold`.
- Using a `@tag` and a `@filter` that references the tag within the same vertex is allowed, so long as the two do not appear on the exact same property field.

@filter

Allows filtering of the data to be returned, based on any of a set of filtering operations. Conceptually, it is the GraphQL equivalent of the SQL `WHERE` keyword.

See *Supported filtering operations* for details on the various types of filtering that the compiler currently supports. These operations are currently hardcoded in the compiler; in the future, we may enable the addition of custom filtering operations via compiler plugins.

Multiple `@filter` directives may be applied to the same field at once. Conceptually, it is as if the different `@filter` directives were joined by SQL `AND` keywords.

Using a `@tag` and a `@filter` that references the tag within the same vertex is allowed, so long as the two do not appear on the exact same property field.

Passing Parameters

The `@filter` directive accepts two types of parameters: runtime parameters and tagged parameters.

Runtime parameters are represented with a `$` prefix (e.g. `$foo`), and denote parameters whose values will be known at runtime. The compiler will compile the GraphQL query leaving a spot for the value to fill at runtime. After compilation, the user will have to supply values for all runtime parameters, and their values will be inserted into the final query before it can be executed against the database.

Consider the following query:

```
{
    Animal {
        name @output(out_name: "animal_name")
        color @filter(op_name: "=", value: ["$animal_color"])
    }
}
```

It returns one row for every `Animal` vertex that has a color equal to `$animal_color`. Each row contains the animal's name in a column named `animal_name`. The parameter `$animal_color` is a runtime parameter – the user must pass in a value (e.g. `{"animal_color": "blue"}`) that will be inserted into the query before querying the database.

Tagged parameters are represented with a `%` prefix (e.g. `%foo`) and denote parameters whose values are derived from a property field encountered elsewhere in the query. If the user marks a property field with a `@tag` directive and a suitable name, that value becomes available to use as a tagged parameter in all subsequent `@filter` directives.

Consider the following query:

```
{
    Animal {
        name @tag(out_name: "parent_name")
        out_Animal_ParentOf {
```

(continues on next page)

(continued from previous page)

```
            name @filter(op_name: "has_substring", value: ["%parent_name"])
                 @output(out_name: "child_name")
        }
    }
}
```

It returns the names of animals that contain their parent's name as a substring of their own. The database captures the value of the parent animal's name as the `parent_name` tag, and this value is then used as the `%parent_name` tagged parameter in the child animal's `@filter`.

We considered and **rejected** the idea of allowing literal values (e.g. `123`) as `@filter` parameters, for several reasons:

- The GraphQL type of the `@filter` directive's `value` field cannot reasonably encompass all the different types of arguments that people might supply. Even counting scalar types only, there's already `ID, Int, Float, Boolean, String, Date, DateTime...` – way too many to include.
- Literal values would be used when the parameter's value is known to be fixed. We can just as easily accomplish the same thing by using a runtime parameter with a fixed value. That approach has the added benefit of potentially reducing the number of different queries that have to be compiled: two queries with different literal values would have to be compiled twice, whereas using two different sets of runtime arguments only requires the compilation of one query.
- We were concerned about the potential for accidental misuse of literal values. SQL systems have supported stored procedures and parameterized queries for decades, and yet ad-hoc SQL query construction via simple string interpolation is still a serious problem and is the source of many SQL injection vulnerabilities. We felt that disallowing literal values in the query will drastically reduce both the use and the risks of unsafe string interpolation, at an acceptable cost.

Constraints and Rules

- The value provided for `op_name` may only consist of upper or lower case letters (`A-Z`, `a-z`), or underscores (_).
- Values provided in the `value` list must start with either `$` (denoting a runtime parameter) or `%` (denoting a tagged parameter), followed by exclusively upper or lower case letters (`A-Z`, `a-z`) or underscores (_).
- The `@tag` directives corresponding to any tagged parameters in a given `@filter` query must be applied to fields that appear either at the same vertex as the one with the `@filter`, or strictly before the field with the `@filter` directive.
- "Can't compare apples and oranges" – the GraphQL type of the parameters supplied to the `@filter` must match the GraphQL types the compiler infers based on the field the `@filter` is applied to.
- If the `@tag` corresponding to a tagged parameter originates from within a vertex field marked `@optional`, the emitted code for the `@filter` checks if the `@optional` field was assigned a value. If no value was assigned to the `@optional` field, comparisons against the tagged parameter from within that field return `True`.
  - For example, assuming `%from_optional` originates from an `@optional` scope, when no value is assigned to the `@optional` field:
    * using `@filter(op_name: "=", value: ["%from_optional"])` is equivalent to not having the filter at all;
    * using `@filter(op_name: "between", value: ["$lower", "%from_optional"])` is equivalent to `@filter(op_name: ">=", value: ["$lower"])`.

- Using a `@tag` and a `@filter` that references the tag within the same vertex is allowed, so long as the two do not appear on the exact same property field.

@recurse

Applied to a vertex field, specifies that the edge connecting that vertex field to the current vertex should be visited repeatedly, up to `depth` times. The recursion always starts at `depth = 0`, i.e. the current vertex – see the below sections for a more thorough explanation.

Example Use

Say the user wants to fetch the names of the children and grandchildren of each `Animal`. That could be accomplished by running the following two queries and concatenating their results:

```
{
    Animal {
        name @output(out_name: "ancestor")
        out_Animal_ParentOf {
            name @output(out_name: "descendant")
        }
    }
}
```

```
{
    Animal {
        name @output(out_name: "ancestor")
        out_Animal_ParentOf {
            out_Animal_ParentOf {
                name @output(out_name: "descendant")
            }
        }
    }
}
```

If the user then wanted to also add great-grandchildren to the `descendants` output, that would require yet another query, and so on. Instead of concatenating the results of multiple queries, the user can simply use the `@recurse` directive. The following query returns the child and grandchild descendants:

```
{
    Animal {
        name @output(out_name: "ancestor")
        out_Animal_ParentOf {
            out_Animal_ParentOf @recurse(depth: 1) {
                name @output(out_name: "descendant")
            }
        }
    }
}
```

Each row returned by this query contains the name of an `Animal` in the `ancestor` column and the name of its child or grandchild in the `descendant` column. The `out_Animal_ParentOf` vertex field marked `@recurse` is already enclosed within another `out_Animal_ParentOf` vertex field, so the recursion starts at the "child" level (the `out_Animal_ParentOf` not marked with `@recurse`). Therefore, the `descendant` column contains the names of an `ancestor`'s children (from `depth = 0` of the recursion) and the names of its grandchildren (from `depth = 1`).

Recursion using this directive is possible since the types of the enclosing scope and the recursion scope work out: the `@recurse` directive is applied to a vertex field of type `Animal` and its vertex field is enclosed within a scope of type `Animal`. Additional cases where recursion is allowed are described in detail below.

The `descendant` column cannot have the name of the `ancestor` animal since the `@recurse` is already within one `out_Animal_ParentOf` and not at the root `Animal` vertex field. Similarly, it cannot have descendants that are more than two steps removed (e.g., great-grandchildren), since the `depth` parameter of `@recurse` is set to `1`.

Now, let's see what happens when we eliminate the outer `out_Animal_ParentOf` vertex field and simply have the `@recurse` applied on the `out_Animal_ParentOf` in the root vertex field scope:

```
{
    Animal {
        name @output(out_name: "ancestor")
        out_Animal_ParentOf @recurse(depth: 1) {
            name @output(out_name: "self_or_descendant")
        }
    }
}
```

In this case, when the recursion starts at `depth = 0`, the `Animal` within the recursion scope will be the same `Animal` at the root vertex field, and therefore, in the `depth = 0` step of the recursion, the value of the `self_or_descendant` field will be equal to the value of the `ancestor` field.

Constraints and Rules

- "The types must work out" – when applied within a scope of type `A`, to a vertex field of type `B`, at least one of the following must be true:
  - `A` is a GraphQL union;
  - `B` is a GraphQL interface, and `A` is a type that implements that interface;
  - `A` and `B` are the same type.
- `@recurse` can only be applied to vertex fields other than the root vertex field of a query.
- Cannot be used within a scope marked `@optional` or `@fold`.
- The `depth` parameter of the recursion must always have a value greater than or equal to 1. Using `depth = 1` produces the current vertex and its neighboring vertices along the specified edge.
- Type coercions and `@filter` directives within a scope marked `@recurse` do not limit the recursion depth. Conceptually, recursion to the specified depth happens first, and then type coercions and `@filter` directives eliminate some of the locations reached by the recursion.
- As demonstrated by the examples above, the recursion always starts at depth 0, so the recursion scope always includes the vertex at the scope that encloses the vertex field marked `@recurse`.

@output_source

`@output_source` is a directive that can be used on the last vertex field in a query to reverse the order in which vertex fields are visited. Currently, its primary function is to help deal with the following known issue that occurs when compiling to `gremlin`. When vertex fields are visited in a certain order in a GraphQL query, the compiler returns a *gremlin* that does not return the complete set of results promised by the query semantics. See the example use section for more details.

Example use

```
a   ---->_ x
|____   /|
    _|_/
   / |____
  /      \/
b   ----> y
```

Let `a, b, x, y` be the values of the `name` property field of four vertices. Let the vertices named `a` and `b` be of type `S`, and let `x` and `y` be of type `T`. Let vertex `a` be connected to both `x` and `y` via directed edges of type `E`. Similarly, let vertex `b` also be connected to both `x` and `y` via directed edges of type `E`.

Consider the GraphQL query:

```
{
    S {
        name @output(out_name: "s_name")
        out_E {
            name @output(out_name: "t_name")
        }
    }
}
```

Between the data in the database and the query's structure, it is clear that combining any of `a` or `b` with any of `x` or `y` would produce a valid result. Therefore, the complete result list, shown here in JSON format, would be:

```
[
    {"s_name": "a", "t_name": "x"},
    {"s_name": "a", "t_name": "y"},
    {"s_name": "b", "t_name": "x"},
    {"s_name": "b", "t_name": "y"},
]
```

This is precisely what the `MATCH` compilation target is guaranteed to produce. (`MATCH` is our name for the SQL dialect that OrientDB uses). However, the `gremlin` compilation target does not guarantee a complete result list. Querying the database using a query string generated by the `gremlin` compilation target will produce only a partial result list resembling the following:

```
[
    {"s_name": "a", "t_name": "x"},
    {"s_name": "b", "t_name": "x"},
]
```

Due to limitations in the underlying query language, `gremlin` will by default produce at most one result for each of the starting locations in the query. The above GraphQL query started at the type `S`, so each `s_name` in the returned result list is therefore distinct. Furthermore, there is no guarantee (and no way to know ahead of time) whether `x` or `y` will be returned as the `t_name` value in each result, as they are both valid results.

Users may apply the `@output_source` directive on the last scope of the query to alter this behavior:

```
{
    S {
        name @output(out_name: "s_name")
        out_E @output_source {
            name @output(out_name: "t_name")
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Rather than producing at most one result for each `S`, the query will now produce at most one result for each distinct value that can be found at `out_E`, where the directive is applied:

```
[
    {"s_name": "a", "t_name": "x"},
    {"s_name": "a", "t_name": "y"},
]
```

Conceptually, applying the `@output_source` directive makes it as if the query were written in the opposite order:

```
{
    T {
        name @output(out_name: "t_name")
        in_E {
            name @output(out_name: "s_name")
        }
    }
}
```

Constraints and Rules

- May exist at most once in any given GraphQL query.
- Can exist only on a vertex field, and only on the last vertex field used in the query.
- Cannot be used within a scope marked `@optional` or `@fold`.

Supported filtering operations

Comparison operators

Supported comparison operators:

- Equal to: `=`
- Not equal to: `!=`
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`

Example Use

Equal to (=):

```
{
    Species {
        name @filter(op_name: "=", value: ["$species_name"])
        uuid @output(out_name: "species_uuid")
    }
}
```

This returns one row for every `Species` whose name is equal to the value of the `$species_name` parameter. Each row contains the `uuid` of the `Species` in a column named `species_uuid`.

Greater than or equal to (>=):

```
{
    Animal {
        name @output(out_name: "name")
        birthday @output(out_name: "birthday")
                 @filter(op_name: ">=", value: ["$point_in_time"])
    }
}
```

This returns one row for every `Animal` vertex that was born after or on a `$point_in_time`. Each row contains the animal's name and birthday in columns named `name` and `birthday`, respectively.

Constraints and Rules

- All comparison operators must be on a property field.

name_or_alias

Allows you to filter on vertices which contain the exact string `$wanted_name_or_alias` in their `name` or `alias` fields.

Example Use

```
{
    Animal @filter(op_name: "name_or_alias", value: ["$wanted_name_or_alias"]) {
        name @output(out_name: "name")
    }
}
```

This returns one row for every `Animal` vertex whose name and/or alias is equal to `$wanted_name_or_alias`. Each row contains the animal's name in a column named `name`.

The value provided for `$wanted_name_or_alias` must be the full name and/or alias of the `Animal`. Substrings will not be matched.

Constraints and Rules

- Must be on a vertex field that has `name` and `alias` properties.

between

Example Use

```
{
    Animal {
        name @output(out_name: "name")
        birthday @filter(op_name: "between", value: ["$lower", "$upper"])
                 @output(out_name: "birthday")
    }
}
```

This returns:

- One row for every `Animal` vertex whose birthday is in between `$lower` and `$upper` dates (inclusive). Each row contains the animal's name in a column named `name`.

Constraints and Rules

- Must be on a property field.
- The lower and upper bounds represent an inclusive interval, which means that the output may contain values that match them exactly.

in_collection

Example Use

```
{
    Animal {
        name @output(out_name: "animal_name")
        color @output(out_name: "color")
              @filter(op_name: "in_collection", value: ["$colors"])
    }
}
```

This returns one row for every `Animal` vertex which has a color contained in a list of colors. Each row contains the `Animal`'s name and color in columns named `animal_name` and `color`, respectively.

Constraints and Rules

- Must be on a property field that is not of list type.

not_in_collection

Example Use

```
{
    Animal {
        name @output(out_name: "animal_name")
        color @output(out_name: "color")
              @filter(op_name: "not_in_collection", value: ["$colors"])
    }
}
```

This returns one row for every `Animal` vertex which has a color not contained in a list of colors. Each row contains the `Animal`'s name and color in columns named `animal_name` and `color`, respectively.

Constraints and Rules

- Must be on a property field that is not of list type.

has_substring

Example Use

```
{
    Animal {
        name @filter(op_name: "has_substring", value: ["$substring"])
             @output(out_name: "animal_name")
    }
}
```

This returns one row for every `Animal` vertex whose name contains the value supplied for the `$substring` parameter. Each row contains the matching `Animal`'s name in a column named `animal_name`.

Constraints and Rules

- Must be on a property field of string type.

starts_with

Example Use

```
{
    Animal {
        name @filter(op_name: "starts_with", value: ["$prefix"])
             @output(out_name: "animal_name")
    }
}
```

This returns one row for every `Animal` vertex whose name starts with the value supplied for the `$prefix` parameter. Each row contains the matching `Animal`'s name in a column named `animal_name`.

Constraints and Rules

- Must be on a property field of string type.

ends_with

Example Use

```
{
    Animal {
        name @filter(op_name: "ends_with", value: ["$suffix"])
             @output(out_name: "animal_name")
    }
}
```

This returns one row for every `Animal` vertex whose name ends with the value supplied for the `$suffix` parameter. Each row contains the matching `Animal`'s name in a column named `animal_name`.

Constraints and Rules

- Must be on a property field of string type.

contains

Example Use

```
{
    Animal {
        alias @filter(op_name: "contains", value: ["$wanted"])
        name @output(out_name: "animal_name")
    }
}
```

This returns one row for every `Animal` vertex whose list of aliases contains the value supplied for the `$wanted` parameter. Each row contains the matching `Animal`'s name in a column named `animal_name`.

Constraints and Rules

- Must be on a property field of list type.

not_contains

Example Use

```
{
    Animal {
        alias @filter(op_name: "not_contains", value: ["$wanted"])
```

(continues on next page)

(continued from previous page)

```
        name @output(out_name: "animal_name")
    }
}
```

This returns one row for every `Animal` vertex whose list of aliases does not contain the value supplied for the `$wanted` parameter. Each row contains the matching `Animal`'s name in a column named `animal_name`.

Constraints and Rules

- Must be on a property field of list type.

intersects

Example Use

```
{
    Animal {
        alias @filter(op_name: "intersects", value: ["$wanted"])
        name @output(out_name: "animal_name")
    }
}
```

This returns one row for every `Animal` vertex whose list of aliases has a non-empty intersection with the list of values supplied for the `$wanted` parameter. Each row contains the matching `Animal`'s name in a column named `animal_name`.

Constraints and Rules

- Must be on a property field of list type.

has_edge_degree

Example Use

```
{
    Animal {
        name @output(out_name: "animal_name")
        out_Animal_ParentOf @filter(op_name: "has_edge_degree", value: ["$child_count
↪"]) @optional {
            uuid
        }
    }
}
```

This returns one row for every `Animal` vertex that has exactly `$child_count` children (i.e. where the `out_Animal_ParentOf` edge appears exactly `$child_count` times). Each row contains the matching `Animal`'s name, in a column named `animal_name`.

The `uuid` field within the `out_Animal_ParentOf` vertex field is added simply to satisfy the GraphQL syntax rule that requires at least one field to exist within any `{}`. Since this field is not marked with any directive, it has no effect on the query.

N.B.: Please note the `@optional` directive on the vertex field being filtered above. If in your use case you expect to set `$child_count` to 0, you must also mark that vertex field `@optional`. Recall that absence of `@optional` implies that at least one such edge must exist. If the `has_edge_degree` filter is used with a parameter set to 0, that requires the edge to not exist. Therefore, if the `@optional` is not present in this situation, no valid result sets can be produced, and the resulting query will return no results.

Constraints and Rules

- Must be on a vertex field that is not the root vertex of the query.
- Tagged values are not supported as parameters for this filter.
- If the runtime parameter for this operator can be `0`, it is *strongly recommended* to also apply `@optional` to the vertex field being filtered (see N.B. above for details).

is_null

Example Use

```
{
    Animal {
        name @output(out_name: "animal_name")
        color @filter(op_name: "is_null")
    }
}
```

This returns one row for every `Animal` that does not have a color defined.

Constraints and Rules

- Must be applied on a property field.
- `value` must either not appear in the filter (shown in the example) or be an empty list.

is_not_null

Example Use

```
{
    Animal {
        name @output(out_name: "animal_name")
        color @filter(op_name: "is_not_null")
    }
}
```

This returns one row for every `Animal` that has a color defined.

Constraints and Rules

- Must be applied on a property field.
- `value` must either not appear in the filter (shown in the example) or be an empty list.

2.2 Supported Databases

Refer to this section to learn how the compiler integrates with the target database. The database home pages include an end-to-end example, instruction for schema info generation, and any limitations or intricacies related to working with said database. We currently support two types of database backends:

- *OrientDB*
- *SQL Databases*, including SQL Server, Postgres and more.
- *Neo4j/Redisgraph*

2.2.1 OrientDB

The best way to integrate the compiler with OrientDB is by compiling to MATCH, our name for the SQL dialect that OrientDB uses. All query directives are supported when compiling to MATCH. Additionally, since OrientDB is a graph database, generating a GraphQL schema from an OrientDB database requires minimal configuration.

Important: We currently support OrientDB version 2.2.28+.

End-to-End Example

See *Getting Started* for an end-to-end OrientDB example.

Performance Penalties

Compound `optional` Performance Penalty

When compiling to MATCH, including an optional statement in GraphQL has no performance issues on its own, but if you continue expanding vertex fields within an optional scope, there may be significant performance implications.

Going forward, we will refer to two different kinds of `@optional` directives.

- A *"simple"* optional is a vertex with an `@optional` directive that does not expand any vertex fields within it. For example:

```
{
    Animal {
        name @output(out_name: "name")
        in_Animal_ParentOf @optional {
            name @output(out_name: "parent_name")
        }
    }
}
```

OrientDB `MATCH` currently allows the last step in any traversal to be optional. Therefore, the equivalent `MATCH` traversal for the above `GraphQL` is as follows:

```
SELECT
Animal___1.name as `name`,
Animal__in_Animal_ParentOf___1.name as `parent_name`
FROM (
MATCH {
    class: Animal,
    as: Animal___1
}.in('Animal_ParentOf') {
    as: Animal__in_Animal_ParentOf___1
}
RETURN $matches
)
```

- A *"compound"* optional is a vertex with an `@optional` directive which does expand vertex fields within it. For example:

```
{
    Animal {
        name @output(out_name: "name")
        in_Animal_ParentOf @optional {
            name @output(out_name: "parent_name")
            in_Animal_ParentOf {
                name @output(out_name: "grandparent_name")
            }
        }
    }
}
```

Currently, this cannot represented by a simple `MATCH` query. Specifically, the following is *NOT* a valid `MATCH` statement, because the optional traversal follows another edge:

```
-- NOT A VALID QUERY
SELECT
Animal___1.name as `name`,
Animal__in_Animal_ParentOf___1.name as `parent_name`
FROM (
MATCH {
    class: Animal,
    as: Animal___1
}.in('Animal_ParentOf') {
    optional: true,
    as: Animal__in_Animal_ParentOf___1
}.in('Animal_ParentOf') {
    as: Animal__in_Animal_ParentOf__in_Animal_ParentOf___1
}
RETURN $matches
)
```

Instead, we represent a *compound* optional by taking an union (`UNIONALL`) of two distinct `MATCH` queries. For instance, the `GraphQL` query above can be represented as follows:

```
SELECT EXPAND($final_match)
LET
    $match1 = (
        SELECT
```

(continues on next page)

(continued from previous page)

```
            Animal___1.name AS `name`
        FROM (
            MATCH {
                class: Animal,
                as: Animal___1,
                where: (
                    (in_Animal_ParentOf IS null)
                    OR
                    (in_Animal_ParentOf.size() = 0)
                ),
            }
        )
    ),
    $match2 = (
        SELECT
            Animal___1.name AS `name`,
            Animal__in_Animal_ParentOf___1.name AS `parent_name`
        FROM (
            MATCH {
                class: Animal,
                as: Animal___1
            }.in('Animal_ParentOf') {
                as: Animal__in_Animal_ParentOf___1
            }.in('Animal_ParentOf') {
                as: Animal__in_Animal_ParentOf__in_Animal_ParentOf___1
            }
        )
    ),
    $final_match = UNIONALL($match1, $match2)
```

In the first case where the optional edge is not followed, we have to explicitly filter out all vertices where the edge *could have been followed*. This is to eliminate duplicates between the two `MATCH` selections.

Note: The previous example is not *exactly* how we implement *compound* optionals (we also have `SELECT` statements within `$match1` and `$match2`), but it illustrates the the general idea.

Performance Analysis

If we have many *compound* optionals in the given `GraphQL`, the above procedure results in the union of a large number of `MATCH` queries. Specifically, for `n` compound optionals, we generate 2n different `MATCH` queries. For each of the 2n subsets `S` of the `n` optional edges:

- We remove the `@optional` restriction for each traversal in `S`.
- For each traverse `t` in the complement of `S`, we entirely discard `t` along with all the vertices and directives within it, and we add a filter on the previous traverse to ensure that the edge corresponding to `t` does not exist.

Therefore, we get a performance penalty that grows exponentially with the number of *compound* optional edges. This is important to keep in mind when writing queries with many optional directives.

If some of those *compound* optionals contain `@optional` vertex fields of their own, the performance penalty grows since we have to account for all possible subsets of `@optional` statements that can be satisfied simultaneously.

2.2.2 SQL

Relational databases are supported by compiling to SQLAlchemy core as an intermediate language, and then relying on SQLAlchemy's compilation of the dialect-specific SQL query. The compiler does not return a string for SQL compilation, but instead a SQLAlchemy `Query` object that can be executed through a SQLAlchemy engine.

Our SQL backend supports basic traversals, filters, tags and outputs, but there are still some pieces in development:

- Directives: `@fold`
- Filter operators: `has_edge_degree`
- Dialect-specific features, like Postgres array types, and use of filter operators specific to them: `contains`, `intersects`, `name_or_alias`
- Meta fields: `__typename`, `_x_count`

End-to-End SQL Example

To query a SQL backend simply reflect the needed schema data from the database using SQLAlchemy, compile the GraphQL query to a SQLAlchemy `Query`, and execute the query against the engine as in the example below:

```python
from graphql_compiler import get_sqlalchemy_schema_info, graphql_to_sql
from sqlalchemy import MetaData, create_engine

engine = create_engine('<connection string>')

# Reflect the default database schema. Each table must have a primary key.
# See "Including tables without explicitly enforced primary keys" otherwise.
metadata = MetaData(bind=engine)
metadata.reflect()

# Wrap the schema information into a SQLAlchemySchemaInfo object.
sql_schema_info = get_sqlalchemy_schema_info(metadata.tables, {}, engine.dialect)

# Write GraphQL query.
graphql_query = '''
{
    Animal {
        name @output(out_name: "animal_name")
    }
}
'''
parameters = {}

# Compile and execute query.
compilation_result = graphql_to_sql(sql_schema_info, graphql_query, parameters)
query_results = [dict(row) for row in engine.execute(compilation_result.query)]
```

Advanced Features

SQL Edges

Edges can be specified in SQL through the `direct_edges` parameter as illustrated below. SQL edges gets rendered as `out_edgeName` and `in_edgeName` in the source and destination GraphQL objects respectively and edge traversals get compiled to SQL joins between the source and destination tables using the specified columns. We use the term `direct_edges` below since the compiler may support other types of SQL edges in the future such as edges that are backed by SQL association tables.

```
from graphql_compiler import get_sqlalchemy_schema_info, graphql_to_sql
from graphql_compiler.schema_generation.sqlalchemy.edge_descriptors import␣
↪DirectEdgeDescriptor
from sqlalchemy import MetaData, create_engine

# Set engine and reflect database metadata. (See example above for more details).
engine = create_engine('<connection string>')
metadata = MetaData(bind=engine)
metadata.reflect()

# Specify SQL edges.
direct_edges = {
    'Animal_LivesIn': DirectEdgeDescriptor(
        from_vertex='Animal',  # Name of the source GraphQL object as specified.
        from_column='location',  # Name of the column of the underlying source table␣
↪to join on.
        to_vertex='Location',  # Name of the destination GraphQL object as specified.
        to_column='uuid',   # Name of the column of the underlying destination table␣
↪to join on.
     )
}

# Wrap the schema information into a SQLAlchemySchemaInfo object.
sql_schema_info = get_sqlalchemy_schema_info(metadata.tables, direct_edges, engine.
↪dialect)

# Write GraphQL query with edge traversal.
graphql_query = '''
{
    Animal {
        name @output(out_name: "animal_name")
        out_Animal_LivesIn {
            name @output(out_name: "location_name")
        }
    }
}
'''

# Compile query. Note that the edge traversal gets compiled to a SQL join.
compilation_result = graphql_to_sql(sql_schema_info, graphql_query, {})
```

Including tables without explicitly enforced primary keys

The compiler requires that each SQLAlchemy `Table` object in the `SQLALchemySchemaInfo` has a primary key. However, the primary key in the `Table` need not be the primary key in the underlying table. It may simply be a non-null and unique identifier of each row. To override the primary key of SQLAlchemy `Table` objects reflected from a database please follow the instructions in this link.

Including tables from multiple schemas

SQLAlchemy and SQL database management systems support the concept of multiple schemas. One can include `Table` objects from multiple schemas in the same `SQLAlchemySchemaInfo`. However, when doing so, one cannot simply use table names as GraphQL object names because two tables in different schemas can have the same the name. A solution that is not quite guaranteed to work, but will likely work in practice is to prepend the schema name as follows:

```
vertex_name_to_table = {}
for table in metadata.values():
    # The schema field may be None if the database name is specified in the␣
↪connection string
    # and the table is in the default schema, (e.g. 'dbo' for mssql and 'public' for␣
↪postgres).
    if table.schema:
        vertex_name = 'dbo' + table.name
    else:
        # If the database name is not specified in the connection string, then
        # the schema field is of the form <databaseName>.<schemaName>.
        # Since dots are not allowed in GraphQL type names we must remove them here.
        vertex_name = table.schema.replace('.', '') + table.name

    if vertex_name in vertex_name_to_table:
        raise AssertionError('Found two tables with conflicting GraphQL object names.
↪')

    vertex_name_to_table[vertex_name] = table
```

Including manually defined `Table` objects

The `Table` objects in the `SQLAlchemySchemaInfo` do not need to be reflected from the database. They also can be manually specified as in this link. However, if specifying `Table` objects manually, please make sure to include a primary key for each table and to use only SQL types allowed for the dialect specified in the `SQLAlchemySchemaInfo`.

2.2.3 Neo4j/Redisgraph

Cypher query parameters

RedisGraph doesn't support query parameters, so we perform manual parameter interpolation in the `graphql_to_redisgraph_cypher` function. However, for Neo4j, we can use Neo4j's client to do parameter interpolation on its own so that we don't reinvent the wheel.

The function `insert_arguments_into_query` does so based on the query language, which isn't fine-grained enough here– for Cypher backends, we only want to insert parameters if the backend is RedisGraph, but not if it's Neo4j.

Instead, the correct approach for Neo4j Cypher is as follows, given a Neo4j Python client called `neo4j_client`:

```
common_schema_info = CommonSchemaInfo(schema, type_equivalence_hints)
compilation_result = compile_graphql_to_cypher(common_schema_info, graphql_query)
with neo4j_client.driver.session() as session:
    result = session.run(compilation_result.query, parameters)
```

2.3 Advanced Features

To learn more about the advanced features in the GraphQL compiler see:

- *Macro System* to learn how to write "macro edges", which allow users to define new edges that become part of the GraphQL schema, using existing edges as building blocks.
- *Schema Graph* for an utility that makes it easy to explore the schema of a database, including the databases indexes.
- *Additional Tools* for a list of additional tools included in the package, including a query pretty printer.

2.3.1 Macro System

The macro system allows users to reshape how they *perceive* their data, without requiring changes to the underlying database structures themselves.

In many real-life situations, the database schema does not fit the user's mental model of the data. There are many causes of this, the most common one being database normalization. The representation of the data that is convenient for storage within a database is rarely the representation that makes for easy querying. As a result, users' queries frequently include complex and repetitive query structures that work around the database's chosen data model.

The compiler's macro system empowers users *reshaping* their data's structure to fit their mental model, minimizing query complexity and repetitiveness without requiring changes to the shape of the data in the underlying data systems. The compiler achieves this by allowing users to define **macros** – type-safe rules for programmatic query rewriting that transform user-provided queries on the *desired* data model into queries on the *actual* data model in the underlying data systems.

When macros are defined, the compiler loads them into a *macro registry* – a data structure that tracks all currently available macros, the resulting GraphQL schema (accounting for macros), and any additional metadata needed by the compiler. The compiler then leverages this registry to expand queries that rely on macros, rewriting them into equivalent queries that do not contain any macros and therefore reflect the actual underlying data model.

This makes macros somewhat similar to SQL's idea of non-materialized views, though there are some key differences:

- SQL views require database access and special permissions; databases are completely oblivious to the use of macros since by the time the database gets the query, all macro uses have been already expanded.
- Macros can be stored and expanded client-side, so different users that query the same system may define their own personal macros which are not shared with other users or the server that executes the users' GraphQL queries. This is generally not achievable with SQL.
- Since macro expansion does not interact in any way with the underlying data system, it works seamlessly with all databases and even on schemas stitched together from multiple databases. In contrast, not all databases support SQL-like `VIEW` functionality.

Currently, the compiler supports one type of macro: *macro edges*, which allow the creation of "virtual" edges computed from existing ones. More types of macros are coming in the future.

Macro registry

The macro registry is where the definitions of all currently defined macros are stored, together with the resulting GraphQL schema they form, as well as any associated metadata that the compiler's macro system may need in order to expand any macros encountered in a query.

To create a macro registry object for a given GraphQL schema, use the `create_macro_registry` function:

```
from graphql_compiler.macros import create_macro_registry

macro_registry = create_macro_registry(your_graphql_schema_object)
```

To retrieve the GraphQL schema object with all its macro-based additions, use the `get_schema_with_macros` function:

```
from graphql_compiler.macros import get_schema_with_macros

graphql_schema = get_schema_with_macros(macro_registry)
```

Schema for defining macros

Macro definitions rely on additional directives that are not normally defined in the schema the GraphQL compiler uses for querying. We intentionally do not include these directives in the schema used for querying, since defining macros and writing queries are different modes of use of the compiler, and we believe that controlling which sets of directives are available in which mode will minimize the potential for user confusion.

The `get_schema_for_macro_definition()` function is able to transform a querying schema into one that is suitable for defining macros. Getting such a schema may be useful, for example, when setting up a GraphQL editor (such as GraphiQL) to create and edit macros.

Macro edges

Macro edges allow users to define new edges that become part of the GraphQL schema, using existing edges as building blocks. They allow users to define shorthand for common querying operations, encapsulating uses of existing query functionality (e.g., tags, filters, recursion, type coercions, etc.) into a virtual edge with a user-specified name that exists only on a specific GraphQL type (and all its subtypes). Both macro edge definitions and their uses are fully type-checked, ensuring the soundness of both the macro definition and any queries that use it.

Overview and use of macro edges

Let us explain the idea of macro edges through a simple example.

Consider the following query, which returns the list of grandchildren of a given animal:

```
{
    Animal {
        name @filter(op_name: "=", value: ["$animal_name"])
        out_Animal_ParentOf {
            out_Animal_ParentOf {
                name @output(out_name: "grandchild_name")
            }
        }
    }
}
```

If operations on animals' grandchildren are common in our use case, we may wish that an edge like `out_Animal_GrandparentOf` had existed and saved us some repetitive typing.

One of our options is to materialize such an edge in the underlying database itself. However, this causes denormalization of the database – there are now two places where an animal's grandchildren are written down – requiring additional storage space, and introducing potential for user confusion and data inconsistency between the two representations.

Another option is to introduce a non-materialized view within the database that *makes it appear* that such an edge exists, and query this view via the GraphQL compiler. While this avoids some of the drawbacks of the previous approach, not all databases support non-materialized views. Also, querying users are not always able to add views to the database, and may require additional permissions on the database system.

Macro edges give us the opportunity to define a new `out_Animal_GrandparentOf` edge without involving the underlying database systems at all. We simply state that such an edge is constructed by composing two `out_Animal_ParentOf` edges together:

```
from graphql_compiler.macros import register_macro_edge

macro_edge_definition = '''{
    Animal @macro_edge_definition(name: "out_Animal_GrandparentOf") {
        out_Animal_ParentOf {
            out_Animal_ParentOf @macro_edge_target {
                uuid
            }
        }
    }
}'''
macro_edge_args = {}

register_macro_edge(your_macro_registry_object, macro_edge_definition, macro_edge_
↪args)
```

Let's dig into the GraphQL macro edge definition one step at a time:

- We know that the new macro edge is being defined on the `Animal` GraphQL type, since that is the type where the definition begins.
- The `@macro_edge_definition` directive specifies the name of the new macro edge.
- The newly-defined `out_Animal_GrandparentOf` edge connects `Animal` vertices to the vertices reachable after exactly two traversals along `out_Animal_ParentOf` edges; this is what the `@macro_edge_target` directive signifies.
- As the `out_Animal_ParentOf` field containing the `@macro_edge_target` directive is of type `[Animal]` (we know this from our schema), the compiler will automatically infer that the `out_Animal_GrandparentOf` macro edge also points to vertices of type `Animal`.
- The `uuid` within the inner `out_Animal_ParentOf` scope is a "pro-forma" field – it is there simply to satisfy the GraphQL parser, since per the GraphQL specification, each pair of curly braces must reference at least one field. The named field has no meaning in this definition, and the user may choose to use any field that exists within that pair of curly braces. The preferred convention for pro-forma fields is to use whichever field represents the primary key of the given type in the underlying database.
- This macro edge does not take arguments, so we set the `macro_edge_args` value to an empty dictionary. We will cover macro edges with arguments later.

Having defined this macro edge, we are now able to rewrite our original query into a simpler yet equivalent form:

```
{
    Animal {
        name @filter(op_name: "=", value: ["$animal_name"])
        out_Animal_GrandparentOf {
            name @output(out_name: "grandchild_name")
        }
    }
}
```

We can now observe the process of macro expansion in action:

```
from graphql_compiler.macros import get_schema_with_macros, perform_macro_expansion

```

(continues on next page)

(continued from previous page)

```
query = '''{
    Animal {
        name @filter(op_name: "=", value: ["$animal_name"])
        out_Animal_GrandparentOf {
            name @output(out_name: "grandchild_name")
        }
    }
}'''
args = {
    'animal_name': 'Hedwig',
}

schema_with_macros = get_schema_with_macros(macro_registry)
new_query, new_args = perform_macro_expansion(macro_registry, schema_with_macros,␣
↪query, args)

print(new_query)
# Prints out the following query:
# {
#     Animal {
#         name @filter(op_name: "=", value: ["$animal_name"])
#         out_Animal_ParentOf {
#             out_Animal_ParentOf {
#                 name @output(out_name: "grandchild_name")
#             }
#         }
#     }
# }

print(new_args)
# Prints out the following arguments:
# {'animal_name': 'Hedwig'}
```

Advanced macro edges use cases

When defining macro edges, one may freely use other compiler query functionality, such as `@recurse`, `@filter`, `@tag`, and so on. Here is a more complex macro edge definition that relies on such more advanced features to define an edge that connects `Animal` vertices to their siblings who are both older and have a higher net worth:

```
from graphql_compiler.macros import register_macro_edge

macro_edge_definition = '''
{
    Animal @macro_edge_definition(name: "out_Animal_RicherOlderSiblings") {
        net_worth @tag(tag_name: "self_net_worth")
        out_Animal_BornAt {
            event_date @tag(tag_name: "self_birthday")
        }
        in_Animal_ParentOf {
            out_Animal_ParentOf @macro_edge_target {
                net_worth @filter(op_name: ">", value: ["%self_net_worth"])
                out_Animal_BornAt {
                    event_date @filter(op_name: "<", value: ["%self_birthday"])
                }
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }
}'''
macro_edge_args = {}

register_macro_edge(your_macro_registry_object, macro_edge_definition, macro_edge_
↪args)
```

Similarly, macro edge definitions are also able to use runtime parameters in their `@filter` directives, by simply including the runtime parameters needed by the macro edge in the call to `register_macro_edge()`. The following example defines a macro edge connecting `Animal` vertices to their grandchildren that go by the name of "Nate".

```
macro_edge_definition = '''
{
    Animal @macro_edge_definition(name: "out_Animal_GrandchildrenCalledNate") {
        out_Animal_ParentOf {
            out_Animal_ParentOf @filter(op_name: "name_or_alias", value: ["$nate_name
↪"])
                                @macro_edge_target {
                uuid
            }
        }
    }
}'''
macro_edge_args = {
    'nate_name': 'Nate',
}

register_macro_edge(your_macro_registry_object, macro_edge_definition, macro_edge_
↪args)
```

When a GraphQL query uses this macro edge, the `perform_macro_expansion()` function will automatically ensure that the macro edge's arguments become part of the expanded query's arguments:

```
query = '''{
    Animal {
        name @output(out_name: "animal_name")
        out_Animal_GrandchildrenCalledNate {
            uuid @output(out_name: "grandchild_id")
        }
    }
}'''
args = {}
schema_with_macros = get_schema_with_macros(macro_registry)
expanded_query, new_args = perform_macro_expansion(
      macro_registry, schema_with_macros, query, args)

print(expanded_query)
# Prints out the following query:
# {
#     Animal {
#         name @output(out_name: "animal_name")
#         out_Animal_ParentOf {
#             out_Animal_ParentOf @filter(op_name: "name_or_alias", value: ["$nate_
↪name"]) {
#                 uuid @output(out_name: "grandchild_id")
```

(continues on next page)

(continued from previous page)

```
#               }
#           }
#       }
#   }

print(new_args)
# Prints out the following arguments:
# {'nate_name': 'Nate'}
```

Constraints and rules for macro edge definitions

- Macro edge definitions cannot use other macros as part of their definition.
- A macro definition contains exactly one `@macro_edge_definition` and one `@macro_edge_target` directive. These directives can only be used within macro edge definitions.
- The `@macro_edge_target` cannot be at or within a scope marked `@fold` or `@optional`.
- The scope marked `@macro_edge_target` cannot immediately contain a type coercion. Instead, place the `@macro_edge_target` directive at the type coercion itself instead of on its enclosing scope.
- Macros edge definitions cannot contain uses of `@output` or `@output_source`.

Constraints and rules for macro edge usage

- The `@optional` and `@recurse` directives cannot be used on macro edges.
- During the process of macro edge expansion, any directives applied on the vertex field belonging to the macro edge are applied to the vertex field marked with `@macro_edge_target` in the macro edge's definition.

In the future, we hope to add support for using `@optional` on macro edges. We have opened a GitHub issue to track this effort, and we welcome contributions!

2.3.2 Schema Graph

When building a GraphQL schema from the database metadata, we first build a `SchemaGraph` from the metadata and then, from the `SchemaGraph`, build the GraphQL schema. The `SchemaGraph` is also a representation of the underlying database schema, but it has three main advantages that make it a more powerful schema introspection tool:

1. It's able to store and expose a schema's index information. The interface for accessing index information is provisional though and might change in the near future.
2. Its classes are allowed to inherit from non-abstract classes.
3. It exposes many utility functions, such as `get_subclass_set`, that make it easier to explore the schema.

See below for a mock example of how to build and use the `SchemaGraph`:

```
from graphql_compiler.schema_generation.orientdb.schema_graph_builder import (
    get_orientdb_schema_graph
)
from graphql_compiler.schema_generation.orientdb.utils import (
    ORIENTDB_INDEX_RECORDS_QUERY, ORIENTDB_SCHEMA_RECORDS_QUERY
)

```

(continues on next page)

(continued from previous page)

```
# Get schema metadata from hypothetical Animals database.
client = your_function_that_returns_a_pyorient_client()
schema_records = client.command(ORIENTDB_SCHEMA_RECORDS_QUERY)
schema_data = [record.oRecordData for record in schema_records]

# Get index data.
index_records = client.command(ORIENTDB_INDEX_RECORDS_QUERY)
index_query_data = [record.oRecordData for record in index_records]

# Build SchemaGraph.
schema_graph = get_orientdb_schema_graph(schema_data, index_query_data)

# Get all the subclasses of a class.
print(schema_graph.get_subclass_set('Animal'))
# {'Animal', 'Dog'}

# Get all the outgoing edge classes of a vertex class.
print(schema_graph.get_vertex_schema_element_or_raise('Animal').out_connections)
# {'Animal_Eats', 'Animal_FedAt', 'Animal_LivesIn'}

# Get the vertex classes allowed as the destination vertex of an edge class.
print(schema_graph.get_edge_schema_element_or_raise('Animal_Eats').out_connections)
# {'Fruit', 'Food'}

# Get the superclass of all classes allowed as the destination vertex of an edge␣
↪class.
print(schema_graph.get_edge_schema_element_or_raise('Animal_Eats').base_out_
↪connection)
# Food

# Get the unique indexes defined on a class.
print(schema_graph.get_unique_indexes_for_class('Animal'))
# [IndexDefinition(name='uuid', 'base_classname'='Animal', fields={'uuid'},␣
↪unique=True, ordered=False, ignore_nulls=False)]
```

We currently support `SchemaGraph` auto-generation for both OrientDB and SQL database backends. In the future, we plan to add a mechanism where one can query a `SchemaGraph` using GraphQL queries.

2.3.3 Additional Tools

GraphQL Query Pretty-Printer

To pretty-print GraphQL queries, use the included pretty-printer:

```
python -m graphql_compiler.tool <input_file.graphql >output_file.graphql
```

It's modeled after Python's `json.tool`, reading from stdin and writing to stdout.

2.4 About GraphQL compiler

To learn more about the GraphQL compiler project see:

- *Contributing* for instructions on how you can contribute.

- *Code of Conduct* for the contributor code of conduct.
- *Changelog* for a history of changes.
- *FAQ* for a list of frequently asked questions.
- *Execution Model* to learn more about the design principles guiding the development of the compiler and the guarantees the compiler provides.

2.4.1 Contributing

Thank you for taking the time to contribute to this project!

To get started, make sure that you have `pipenv`, `docker` and `docker-compose` installed on your computer.

Although GraphQL compiler supports multiple Python 3.6+ versions, we have chosen to use Python 3.8 for development. If you do not already have it installed, consider doing so using pyenv.

If developing on Linux, please also ensure that your Python installation includes header files. The command to install Python header files should look something like this, depending on chosen flavor of Linux. ..

```
sudo apt-get install python3.8-dev
```

Database Driver Installations

Integration tests are run against multiple databases, some of which require that you install specific drivers. Below you'll find the installation instructions for these drivers for Ubuntu and OSX. You might need to run some of the commands with `sudo` depending on your local setup.

MySQL Driver

For MySQL a compatible driver can be installed on OSX with:

```
brew install mysql
```

or on Ubuntu with:

```
apt-get install libmysqlclient-dev python-mysqldb
```

For more details on other systems please refer to MySQL dialect information.

Microsoft SQL Server ODBC Driver

For MSSQL, you can install the required ODBC driver on OSX with:

```
brew tap microsoft/mssql-release https://github.com/Microsoft/homebrew-mssql-release
brew install msodbcsql17 mssql-tools
```

Or Ubuntu with:

```
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
add-apt-repository "$(wget -qO- https://packages.microsoft.com/config/ubuntu/"$(lsb_
↪release -r -s)"/prod.list)"
apt-get update
```

(continues on next page)

(continued from previous page)

```
ACCEPT_EULA=Y apt-get install msodbcsql17
apt-get install unixodbc-dev
```

To see the installation instructions for other operating systems, please follow this link.

Running tests

Once the dev environment is prepared, you can run the tests, from the root repository, with:

```
docker-compose up -d
pipenv sync --dev
pipenv shell

pytest graphql_compiler/tests
```

Some snapshot and integration tests take longer to setup, run, and teardown. These can be optionally skipped during development by running:

```
pytest -m 'not slow'
```

If you run into any issues, please consult the *troubleshooting guide*. If you encounter and resolve an issue that is not already part of the troubleshooting guide, we'd appreciate it if you open a pull request and update the guide to make future development easier.

A test method or class can be marked as slow to be skipped in this fashion by decorating with the `@pytest.mark.slow` flag.

Code of Conduct

This project adheres to the Contributor Covenant code of conduct. By participating, you are expected to uphold this code. Please report unacceptable behavior at graphql-compiler-maintainer@kensho.com.

Contributor License Agreement

Each contributor is required to agree to our Contributor License Agreement, to ensure that their contribution may be safely merged into the project codebase and released under the existing code license. This agreement does not change contributors' rights to use the contributions for any other purpose – it is simply used for the protection of both the contributors and the project.

Style Guide

This project primarily follows the PEP 8 style guide, and secondarily the Google Python style guide. If the style guides differ on a convention, the PEP 8 style guide is preferred.

Additionally, any contributions must pass the linter `scripts/lint.sh` when executed from a pipenv shell (i.e. after running `pipenv shell`). To run the linter on changed files only, commit your changes and run `scripts/lint.sh --diff`. Some linters can automatically fix errors. Use `scripts/fix_lint.sh` to run the automatic fixes.

Finally, all python files in the repository must display the copyright of the project, to protect the terms of the license. Please make sure that your files start with a line like:

```
# Copyright 20xx-present Kensho Technologies, LLC.
```

Read the Docs

To host our documentation we use Read the Docs, a web utility that makes it easy to view and present documentation.

We have taken measures so that the hosted documentation is updated, tested and monitored automatically. We configured a Github webhook so that the hosted documentation is updated every time the main branch gets updated, test the documentation during CI and configured Read the Docs to send notifications to graphql-compiler-maintainer@kensho.com in case there are any issues with building the documentation that were not caught during CI.

Since Read the Docs does not currently support Pipfiles, we must keep the documentation building requirements in both the repository's `Pipfile`, which we use for continuous integration and local development, and in `docs/requirements.txt`, which we use for Read The Docs.

The relevant documentation source code lives in:

```
docs/source
```

To build the website run:

```
pipenv shell
cd docs
make clean
make html
```

Then open `docs/build/index.html` with a web browser to view it.

Troubleshooting Guide

Issues starting MySQL, PostgreSQL, or redis server with docker-compose

If you have any trouble starting the MySQL/PostgreSQL database or the redis server, make sure any database service or any other related service is not already running outside of docker. On OSX, you can stop the MySQL, PostgreSQL, and redis server services by executing:

```
brew services stop mysql
brew services stop postgresql
brew services stop redis-server
```

or on Ubuntu with:

```
service mysql stop
service postgresql stop
service redis-server stop
```

Issues installing the Python MySQL package

Sometimes, precompiled wheels for the Python MySQL package are not available, and your pipenv may try to build the wheels itself. This has happened on OSX and Ubuntu.

OSX

You may then sometimes see an error like the following:

```
[pipenv.exceptions.InstallError]:   File "/usr/local/lib/python3.7/site-packages/
↪pipenv/core.py", line 1874, in do_install
[pipenv.exceptions.InstallError]:       keep_outdated=keep_outdated
[pipenv.exceptions.InstallError]:   File "/usr/local/lib/python3.7/site-packages/
↪pipenv/core.py", line 1253, in do_init
[pipenv.exceptions.InstallError]:       pypi_mirror=pypi_mirror,
[pipenv.exceptions.InstallError]:   File "/usr/local/lib/python3.7/site-packages/
↪pipenv/core.py", line 859, in do_install_dependencies
[pipenv.exceptions.InstallError]:       retry_list, procs, failed_deps_queue,␣
↪requirements_dir, **install_kwargs
[pipenv.exceptions.InstallError]:   File "/usr/local/lib/python3.7/site-packages/
↪pipenv/core.py", line 763, in batch_install
[pipenv.exceptions.InstallError]:       _cleanup_procs(procs, not blocking, failed_
↪deps_queue, retry=retry)
[pipenv.exceptions.InstallError]:   File "/usr/local/lib/python3.7/site-packages/
↪pipenv/core.py", line 681, in _cleanup_procs
[pipenv.exceptions.InstallError]:       raise exceptions.InstallError(c.dep.name,␣
↪extra=err_lines)
[pipenv.exceptions.InstallError]: ['Collecting mysqlclient==1.3.14
...
< lots of error output >
...
ld: library not found for -lssl
...
< lots more error output >
...
error: command 'clang' failed with exit status 1
...
```

The solution is to install OpenSSL on your system:

```
brew install openssl
```

Then, make sure that `clang` is able to find it by adding the following line to your `.bashrc`.

```
export LIBRARY_PATH=$LIBRARY_PATH:/usr/local/opt/openssl/lib/
```

Ubuntu 18.04

When running

```
pipenv install --dev
```

you might get an error like the following:

```
[pipenv.exceptions.InstallError]:   File "/home/$USERNAME/.local/lib/python2.7/site-
↪packages/pipenv/core.py", line 1875, in do_install

[pipenv.exceptions.InstallError]:       keep_outdated=keep_outdated

[pipenv.exceptions.InstallError]:   File "/home/$USERNAME/.local/lib/python2.7/site-
↪packages/pipenv/core.py", line 1253, in do_init
```

(continues on next page)

(continued from previous page)

```
[pipenv.exceptions.InstallError]:       pypi_mirror=pypi_mirror,

[pipenv.exceptions.InstallError]:   File "/home/$USERNAME/.local/lib/python2.7/site-
↪packages/pipenv/core.py", line 859, in do_install_dependencies

[pipenv.exceptions.InstallError]:       retry_list, procs, failed_deps_queue,␣
↪requirements_dir, **install_kwargs

[pipenv.exceptions.InstallError]:   File "/home/$USERNAME/.local/lib/python2.7/site-
↪packages/pipenv/core.py", line 763, in batch_install

[pipenv.exceptions.InstallError]:       _cleanup_procs(procs, not blocking, failed_
↪deps_queue, retry=retry)

[pipenv.exceptions.InstallError]:   File "/home/$USERNAME/.local/lib/python2.7/site-
↪packages/pipenv/core.py", line 681, in _cleanup_procs

[pipenv.exceptions.InstallError]:       raise exceptions.InstallError(c.dep.name,␣
↪extra=err_lines)

[pipenv.exceptions.InstallError]: ['Collecting mysqlclient==1.3.14 (from -r /tmp/
↪pipenv-ZMU3RA-requirements/pipenv-n_utvZ-requirement.txt (line 1))', '  Using␣
↪cached https://files.pythonhosted.org/packages/f7/a2/
↪1230ebbb4b91f42ad6b646e59eb8855559817ad5505d81c1ca2b5a216040/mysqlclient-1.3.14.tar.
↪gz']

[pipenv.exceptions.InstallError]: ['ERROR: Complete output from command python setup.
↪py egg_info:', '    ERROR: /bin/sh: 1: mysql_config: not found', '    Traceback␣
↪(most recent call last):', '      File "<string>", line 1, in <module>', '      ␣
↪File "/tmp/pip-install-ekmq8s3j/mysqlclient/setup.py", line 16, in <module>', '    ␣
↪   metadata, options = get_config()', '      File "/tmp/pip-install-ekmq8s3j/
↪mysqlclient/setup_posix.py", line 53, in get_config', '        libs = mysql_config(
↪"libs_r")', '      File "/tmp/pip-install-ekmq8s3j/mysqlclient/setup_posix.py",␣
↪line 28, in mysql_config', '        raise EnvironmentError("%s not found" % (mysql_
↪config.path,))', '    OSError: mysql_config not found', '    ------------------------
↪----------------', 'ERROR: Command "python setup.py egg_info" failed with error␣
↪code 1 in /tmp/pip-install-ekmq8s3j/mysqlclient/']
```

The solution is to install MySQL:

```
sudo apt-get install python3.8-dev libmysqlclient-dev
```

after which

```
pipenv install --dev
```

should work fine.

This error might happen even if you've run

```
apt-get install python-mysqldb
```

because that only installs the interface to MySQL.

Issues with pyodbc

If you have any issues installing `pydobc` when running `pipenv install`, then it might mean that you have failed to correctly install the ODBC driver.

Another reason that your *pyodbc* installation might fail is because your python installation did not include the required header files. This issue has only affected Ubuntu users so far and can be resolved on Ubuntu by running:

2.4.2 Contributor Covenant Code of Conduct

Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at graphql-compiler-maintainer@kensho.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://contributor-covenant.org/version/2/0

2.4.3 Changelog

Current development version

v2.0.0 (upcoming release)

- **BREAKING** Change the `GraphQLDateTime` scalar type from being timezone-aware to being timezone-naive to follow the usual database convention of naming the timezone-naive type "datetime" and avoid confusion after we've added both timezone-aware and timezone-naive types. #827

v1.11.0

- Release automatic GraphQL schema generation from OrientDB schema metadata. #204
- Release the SchemaGraph, a utility class designed for easy schema introspection. #292
- Release `not_contains` and `not_in_collection` filter operations. #349 #350
- Allow out-of-order `@tag` and `@filter` when in the same scope. #351
- Fix a bug causing MATCH queries to have missing type coercions. #332
- Release functionality that is able to amend parsing and serialization of custom scalar types in schemas parsed from text form. #398
- Improve validation error messages for output and parameter names. #414 #416
- Alpha (unstable) release of query cost estimation functionality. #345
- Clean up README.md and update troubleshooting documentation.
- Many maintainer quality-of-life improvements.

Thanks to `0xflotus`, `bojanserafimov`, `evantey`, `LWProgramming`, `pmantica1`, `qqi0O0`, and `Vlad` for their contributions.

v1.10.1

- Fix `_x_count` and optional filter creating duplicate GlobalOperationsStart IR blocks. #253.
- Raise error for unused `@tag` directives #224.
- Much documentation cleanup and many maintainer quality-of-life improvements.

Thanks to `bojanserafimov`, `evantey14`, `jeremy.meulemans`, and `pmantica1` for their contributions.

v1.10.0

- **BREAKING**: Rename the `__count` meta field to `_x_count`, to avoid GraphQL schema parsing issues with other GraphQL libraries. #176

v1.9.0

- Add a `__count` meta field that supports outputting and filtering on the size of a `@fold` scope. #158
- Add scaffolding for development and testing of SQL compiler backend, and a variety of development quality-of-life improvements.

Thanks to `jmeulemans` for his contributions.

v1.8.3

- Explicit support for Python 3.7. Earlier compiler versions also worked on 3.7, but now we also run tests in 3.7 to confirm. #148
- Bug fix for compilation error when using `has_edge_degree` and `between` filtering in the same scope. #146
- Exposed additional query metadata that describes `@recurse` and `@filter` directives encountered in the query. #141

Thanks to `gurer-kensho` for the contribution.

v1.8.2

- Fix overly strict type check on `@recurse` directives involving a union type. #131

Thanks to `cw6515` for the fix!

v1.8.1

- Fix a bug that arose when using certain type coercions that the compiler optimizes away to a no-op. #127

Thanks to `bojanserafimov` for the fix!

v1.8.0

- Allow `@optional` vertex fields nested inside other `@optional` vertex fields. #120
- Fix a bug that accidentally disallowed having two `@recurse` directives within the same vertex field. #115
- Enforce that all required directives are present in the schema. #114
- Under the hood, made fairly major changes to how query metadata is tracked and processed.

Thanks to `amartyashankha`, `cw6515`, and `yangsong97` for their contributions!

v1.7.2

- Fix possible incorrect query execution due to dropped type coercions. #110 #113

v1.7.0

- Add a new `@filter` operator: `intersects`. #100
- Add an optimization that helps OrientDB choose a good starting point for query evaluation. #102

The new optimization pass manages what type information is visible at different points in the generated query. By exposing additional type information, or hiding existing type information, the compiler maximizes the likelihood that OrientDB will start evaluating the query at the location of lowest cardinality. This produces a massive performance benefit – up to 1000x on some queries!

Thanks to `yangsong97` for making his first contribution with the `intersects` operator!

v1.6.2

- Fix incorrect filtering in `@optional` locations. #95

Thanks to `amartyashankha` for the fix!

v1.6.1

- Fix a bad compilation bug on `@fold` and `@optional` in the same scope. #86

Thanks to `amartyashankha` for the fix!

v1.6.0

- Add full support for `Decimal` data, including both filtering and output. #91

v1.5.0

- Allow expanding vertex fields within `@optional` scopes. #83

This is a massive feature, totaling over 4000 lines of changes and hundreds of hours of many engineers' time. Special thanks to `amartyashankha` for taking point on the implementation!

This feature implements a workaround for a limitation of OrientDB, where `MATCH` treats optional vertices as terminal and does not allow subsequent traversals from them. To work around this issue, the compiler rewrites the query into several disjoint queries whose union produces the exact same results as a single query that allows optional traversals. See this link for more details.

v1.4.1

- Make MATCH use the `BETWEEN` operator when possible, to avoid an OrientDB performance issue #70

Thanks to `amartyashankha` for this contribution!

v1.4.0

- Enable expanding vertex fields inside `@fold` #64

Thanks to `amartyashankha` for this contribution!

v1.3.1

- Add a workaround for a bug in OrientDB related to `@recurse` with type coercions #55
- Exposed the package name and version in the root `__init__.py` file #57

v1.3.0

- Add a new `@filter` operator: `has_edge_degree`. #52
- Lots of under-the-hood cleanup and improvements.

v1.2.1

- Add workaround for OrientDB type inconsistency when filtering lists #42

v1.2.0

- **BREAKING**: Requires OrientDB 2.2.28+, since it depends on two OrientDB bugs being fixed: bug 1 bug 2
- Allow type coercions and filtering within `@fold` scopes.
- Fix bug where `@filter` directives could end up ignored if more than two were in the same scope
- Optimize type coercions in `@optional` and `@recurse` scopes.
- Optimize multiple outputs from the same `@fold` scope.
- Allow having multiple `@filter` directives on the same field #33
- Allow using the `name_or_alias` filtering operation on interface types #37

v1.1.0

- Add support for Python 3 #31
- Make it possible to use `@fold` together with union-typed vertex fields #32

Thanks to `ColCarroll` for making the compiler support Python 3!

v1.0.3

- Fix a minor bug in the GraphQL pretty-printer #30

v1.0.2

- Make the `graphql_to_ir()` easier to use by making it automatically add a new line to the end of the GraphQL query string. Works around an issue in the `graphql-core`dependency library: https://github.com/graphql-python/graphql-core/issues/98
- Robustness improvements for the pretty-printer #27

Thanks to `benlongo` for their contributions.

v1.0.1

- Add GraphQL pretty printer: `python -m graphql_compiler.tool` #23
- Raise errors if there are no `@output` directives within a `@fold` scope #18

Thanks to `benlongo`, `ColCarroll`, and `cw6515` for their contributions.

v1.0.0

Initial release.

Thanks to `MichaelaShtilmanMinkin` for the help in putting the documentation together.

2.4.4 Frequently Asked Questions

Q: Do you really use GraphQL, or do you just use GraphQL-like syntax?

A: We really use GraphQL. Any query that the compiler will accept is entirely valid GraphQL, and we actually use the Python port of the GraphQL core library for parsing and type checking. However, since the database queries produced by compiling GraphQL are subject to the limitations of the database system they run on, our execution model is somewhat different compared to the one described in the standard GraphQL specification.

Q: Does this project come with a GraphQL server implementation?

A: No – there are many existing frameworks for running a web server. We simply built a tool that takes GraphQL query strings (and their parameters) and returns a query string you can use with your database. The compiler does not execute the query string against the database, nor does it deserialize the results. Therefore, it is agnostic to the choice of server framework and database client library used.

Q: Do you plan to support other databases / more GraphQL features in the future?

A: We'd love to, and we could really use your help! Please consider contributing to this project by opening issues, opening pull requests, or participating in discussions.

Q: I think I found a bug, what do I do?

A: Please check if an issue has already been created for the bug, and open a new one if not. Make sure to describe the bug in as much detail as possible, including any stack traces or error messages you may have seen, which database you're using, and what query you compiled.

Q: I think I found a security vulnerability, what do I do?

A: Please reach out to us at graphql-compiler-maintainer@kensho.com so we can triage the issue and take appropriate action.

2.4.5 Execution model

Since the GraphQL compiler can target multiple different query languages, each with its own behaviors and limitations, the execution model must also be defined as a function of the compilation target language. While we strive to minimize the differences between compilation targets, some differences are unavoidable.

The compiler abides by the following principles:

- When the database is queried with a compiled query string, its response must always be in the form of a list of results.
- The precise format of each such result is defined by each compilation target separately.
  - `gremlin`, `MATCH` and `SQL` return data in a tabular format, where each result is a row of the table, and fields marked for output are columns.
  - However, future compilation targets may have a different format. For example, each result may appear in the nested tree format used by the standard GraphQL specification.
- Each such result must satisfy all directives and types in its corresponding GraphQL query.
- The returned list of results is **not** guaranteed to be complete! (This currently only applies to Gremlin - please follow this *link* for more information on the issue).

- In other words, there may have been additional result sets that satisfy all directives and types in the corresponding GraphQL query, but were not returned by the database.
- However, compilation target implementations are encouraged to return complete results if at all practical. The `MATCH` compilation target is guaranteed to produce complete results.