pypika Documentation

Release 3.1.0

Timothy Heys

Oct 14, 2019

Contents

fireant is a a data analysis tool used for quickly building charts, tables, reports, and dashboards. It defines a schema for configuring metrics and dimensions which removes most of the leg work of writing queries and formatting charts. *fireant* even works great with Jupyter notebooks and in the Python shell providing quick and easy access to your data.

CHAPTER 1

Contents

1.1 Installation and Setup

To install *fireant*, run the following command in the terminal:

```
pip install fireant
```

1.1.1 Database Connector add-ons

By default, *fireant* does not include any database drivers. You can optionally include one or provide your own. The following extra installations will set up *fireant* with the recommended drivers. *fireant* may work with additional drivrers, but is untested and requires a custom extension of the `fireant.database.Database` class (see below).

```
# Vertica
pip install fireant[vertica]

# MySQL
pip install fireant[mysql]

# PostgreSQL
pip install fireant[postgresql]

# Amazon Redshift
pip install fireant[redshift]
```

1.1.2 Transformer add-ons

Some transformers have additional dependencies that are not included by default. Include the following extra installations in your *requirements.txt* file if you intend to use those transformers.

```
# matplotlib
pip install fireant[matplotlib]
```

1.2 Database Configuration

1.2.1 Database Connector

In order for *fireant* to connect to your database, a database connector must be used. This takes the form of an instance of a concrete subclass of *fireant*'s `fireant.database.Database` class. Database connectors are shipped with *fireant* for all of the supported databases, but it is also possible to write your own. See below on how to extend *fireant* to support additional databases.

To configure a database, instantiate a subclass of `fireant.database.Database`. You will use this instance to create a *DataSet*. It is possible to use multiple databases simultaneous, but `fireant.DataSet` can only use a single database, since they inherently model the structure of a table in the database.

Vertica

```
import fireant.settings
from fireant.database import VerticaDatabase

database = VerticaDatabase(
    host='example.com',
    port=5433,
    database='example',
    user='user',
    password='password123',
)
```

MySQL

```
import fireant.settings
from fireant.database import MySQLDatabase

database = MySQLDatabase(
    database='testdb',
    host='mysql.example.com',
    port=3308,
    user='user',
    password='password123',
    charset='utf8mb4',
)
```

MySQL additionally requires a custom function that *fireant* uses to rollup date values to specific intervals, equivalent to the `TRUNC_DATE` function available in other database platforms. To install the `TRUNC_DATE` function in your MySQL database, run the script found in `fireant/scripts/mysql_functions.sql`. Further information is provided in this script on how to grant permissions on this function to your MySQL users.

PostgreSQL

```
import fireant.settings
from fireant.database import PostgreSQLDatabase

database = PostgreSQLDatabase(
```

(continues on next page)

(continued from previous page)

```
    database='testdb',
    host='example.com',
    port=5432,
    user='user',
    password='password123',
)
```

Amazon Redshift

```
import fireant.settings
from fireant.database import RedshiftDatabase

fireant.settings = RedshiftDatabase(
    database='testdb',
    host='example.com',
    port=5439,
    user='user',
    password='password123',
)
```

1.2.2 Using a different Database

Instead of using one of the built in database connectors, you can provide your own by extending *fireant.database.Database*.

```
import vertica_python
from pypika import VerticaQuery
from fireant import Database

class MyVertica(Database):
    # Vertica client that uses the vertica_python driver.

    # Override the custom PyPika Query class (Not necessary but perhaps helpful)
    query_cls = VerticaQuery

    def __init__(self, host='localhost', port=5433, database='vertica',
                 user='vertica', password=None,
                 read_timeout=None):
        self.host = host
        self.port = port
        self.database = database
        self.user = user
        self.password = password
        self.read_timeout = read_timeout

    def connect(self):
        return vertica_python.connect(
            host=self.host, port=self.port, database=self.database,
            user=self.user, password=self.password,
            read_timeout=self.read_timeout,
        )

    def trunc_date(self, field, interval):
        return Trunc(...)  # custom Trunc function
```

(continues on next page)

(continued from previous page)

```
    def date_add(self, date_part, interval, field):
        return DateAdd(...)  # custom DateAdd function
```

Once a Database connector has been set up, it can be used when instantiating *fireant.DataSet*.

```
from fireant import DataSet

my_vertica = MyVertica(
    host='example.com',
    port=5433,
    database='example',
    user='user',
    password='password123',
)

DataSet(
    database=my_vertica,
    ...
)
```

In a custom database connector, the `connect` function must be overridden to provide a `connection` to the database. The `trunc_date` and `date_add` functions must also be overridden since are no common ways to truncate/add dates in SQL databases.

1.2.3 Middleware

In order to provide extra functionality as well as flexibility the database connectors allow the setup of middleware. Default configurable middleware implementations are provided by fireant but it's also possible to extend the middleware classes for custom functionality.

Concurrency Middleware

When executing queries on the database the operations are tunneled through a concurrency middleware. By default the *fireant.middleware.ThreadPoolConcurrencyMiddleware* is used when no custom middleware is configured in the database connector. This middleware implementation will parallelize multiple queries using a `ThreadPool`. The maximum amount of simultaneously active threads is then defined by the `max_processes` parameter of the database connector.

A custom middleware can easily be created by implementing *fireant.middleware.BaseConcurrencyMiddleware*. For example a concurrency middleware that would simply execute a group of queries synchronously would look like this:

```
from fireant.middleware import BaseConcurrencyMiddleware
from fireant.queries import fetch_as_dataframe

class HueyConcurrencyMiddleware(BaseConcurrencyMiddleware):
    def fetch_queries_as_dataframe(self, queries, database):
        return [fetch_as_dataframe(query, database) for query in queries]
```

1.3 Creating a *DataSet*

A *DataSet* is a definition of a collection of data that can be queried and transformed into widgets. It consists of four main components: A database connector, a primary database table, join tables, and fields. Once a *DataSet* has been defined, it can be queried to generate a large variety of visualizations.

Some Definitions

Database Connection The database connector is a connection to a database. It contains all of the connection details and supplies the functions for connecting to the database. It also has some helper functions that create SQL where database platforms deviate.

Table The base table to query from in your database. This is the table that goes in the `FROM` clause of the SQL queries generated by *fireant*.

Join Joins specify how to join additional tables. They are instantiated with another PyPika Table and a PyPika expression on how to join the two tables. Joins can also join based on another join by using an expression that links it to the other join table (see below for an example). *fireant* Will automatically determine which joins are necessary on a per query basis.

Field Fields are the bread and butter of *fireant*. The define what types data is available and are ultimately what is referenced when building up queries. Fields are defined with a PyPika expression.

1.3.1 Code Example

```
from fireant.dataset import *
from fireant.database.vertica import VerticaDatabase
from pypika import Tables, functions as fn

vertica_database = VerticaDatabase(user='jane_doe', password='strongpassword123')
analytics, customers = Tables('analytics', 'customers')

dataset = DataSet(
    database=vertica_database,
    table=analytics,
    joins=[
        Join(customers, analytics.customer_id == customers.id),
    ],
    fields=[
        # Non-aggregate definition
        Field(alias='customer',
              definition=customers.id,
              label='Customer'),
        # Date/Time type, also non-aggregate
        Field(alias='date',
              definition=analytics.timestamp,
              type=DataType.date,
              label='Date'),

        # Aggregate definition (The SUM function aggregates a group of values into a␣
↪single value)
        Field(alias='clicks',
              definition=fn.Sum(analytics.clicks),
              label='Clicks'),
    ],
)
```

1.4 Querying with a *DataSet*

After configuring an instance of *fireant.DataSet*, you'll have access to its powerful query API. This follows the builder design pattern, allowing for function calls to be chained to build up a context. The result of a query is one or more widgets, which the query defines as well. Furthermore, data rendered in widgets can be grouped by setting dimensions, filtered, compared over time intervals using references, and sorted and limited using pagination.

Widget A widget is product of executing a *fireant* query. This can come as a Chart or Table, but custom widgets can also be defined.

Dimension Dimensions are references to fields in the context of a *fireant* query. They indicate how the data should be broken up into groups and ultimately expressed as a `GROUP BY` in the SQL query generated by *fireant*.

Filter Filters are expressions that determine which data to include in the result, expressed as either a `WHERE` or `HAVING` clause in the SQL query generated by *fireant*. There are several types of filters and they are in a later section.

Reference A reference is a means to compare data to itself over a time interval. Concretely, these are *Week-over-week* or *Year-over-year* analyses. They are used in a *fireant* query referencing a Field of type `datetime`.

A *DataSet* query uses a dataset's configuration to execute a SQL query and to transform the result into a data visualization, or a widget. A single *fireant* query translates into a single SQL query, but it is possible to transform the a single query into multiple widgets. Queries are constructed with a builder pattern API.

A query is initialized by accessing the `query` attribute on the *fireant.DataSet* instance like `dataset.query`. Subsequent function calls can be chained in order to build up a query.

When fetch is called, the SQL query is executed and the resulting data set is transformed for each widget in the query. The *fetch()* method returns an array containing the data for each widget in the order that the widget was added to the query.

1.4.1 Code Example

```
from fireant.dataset import *
from fireant.database.vertica import VerticaDatabase
from pypika import Tables, functions as fn

query = dataset.query \
    .widget( ... ) \
    .dimension( ... ) \
    .filter( ... ) \
    .reference( ... ) \
    .orderby( ... )

query.fetch()
```

Tip: All builder methods can be called multiple times and in any order.

1.4.2 Builder Functions

Many of the functions in the query builder accept arguments referencing fields defined in the *DataSet*. These can be accessed from your *DataSet* instance via the `fields` property as such:

```
dataset = DataSet(
    ...
    fields=[
        Field(alias='date', ...),
    ]
)

# Access the date field
dataset.fields.date
```

widget Add a widget to a query. A widget is what is returned from the call to `fetch` containing data for a visualization. If a query does not contain a widget, then calling the `fetch` method will return the raw data returned from the database. See *Visualizing the data with Widgets* below for more details.

The function takes one or more arguments that should be instances of subclasses of *fireant.widgets.base.Widget*.

```
from fireant import Pandas

dataset.data \
    ...
   .widget( Pandas(dataset.fields.clicks, dataset.fields.cost, dataset.fields.
↪revenue) )
```

dimension Add a (non-aggregate) field as dimension to the query. This adds a grouping level to the query that will be used to break the data down. See *Grouping data with Dimensions* below for more details.

The function takes one or more arguments of fields in a data set.

```
dataset.data \
    ...
   .dimension( dataset.fields.device, dataset.fields.account )
```

filter Add a filter to the query. This constrains the results of the data by certain criteria. There are many types of filters, see *Filtering the query* below for more details.

The function takes one or more arguments of filter expressions using elements of the dataset that the query is being built from.

```
dataset.data \
    ...
   .filter( dataset.fields.date.between(date(2000, 1, 1), date(2009, 12, 31)) ) \
   .filter( dataset.fields.device.isin(['m', 't']) ) \
   .filter( dataset.fields.clicks > 10 )
```

reference A reference is a way of comparing the data to itself over an interval of time using a Date/Time dimension, such as Week-over-Week. See *Comparing Data to Previous Values using References* below for more details.

```
from fireant import WeekOverWeek

dataset.data \
    ...
   .reference( WeekOverWeek() )
```

orderby This function allows the results of the SQL query to be ordered by fields. Please note that this will *only* order the results of the SQL query and that the order may be affected by the Widget. Ordering is entirely optional. The default order will be by all of the dimensions used in the query in the order that they were added.

```
from pypika import Order

dataset.data \
    ...
   .orderby( dataset.fields.clicks, Order.asc )
```

fetch A call to fetch exits the build function chain and returns the results of the query. An optional hint parameter is accepted which will used in the query if monitoring the queries triggered from *fireant* fireant is needed.

```
from pypika import Order

dataset.data \
    ...
   .fetch()
```

Tip: All of the Data Set query functions accept one or more arguments. Passing in multiple arguments is synonymous as calling the function successively with one argument each.

1.4.3 Grouping data with Dimensions

Dimensions are referenced using the alias defined for them when instantiating the dataset via the `dimensions` attribute on the dataset instance.

```
dataset = dataset(
    ...
    dimensions=[
        UniqueDimension('customer', ... ),
        ...
    ],
    ...
)

# Reference to customer dimension
dataset.fields.customer
```

A dimension can be used in a dataset query by calling the *.dimension(...)* method when building a query. A reference to one or more dimensions must be passed as an argument.

The order of the dimensions is important. The dimensions are grouped in the order that they are added and displayed in the widgets in that order.

```
dataset.data \
    ...
   .dimension( dataset.fields.device, dataset.fields.account )
```

Using intervals with Date/Time dimensions

All continuous dimensions require an interval to group into. For a Date/Time dimension, these intervals are common temporal intervals, such as hours, days, quarters, etc. These dimensions have a default interval and can be used without explicitly setting one. To set the interval, use the reference to the dimension as a function and pass the interval as an arguement

```
from fireant import monthly

dataset.data \
    ...
   .dimension( dataset.fields.date(monthly) )
```

The following intervals are available for Date/Time dimensions and can be imported directly from the `fireant` package.

- `hourly`
- `daily`
- `weekly`
- `monthly`
- `quarterly`
- `annually`

It is also possible to define a custom interval as an instance of *`fireant.DatetimeInterval`*.

Roll Up (Totals)

Rolling up a dimension allows the totals across a dimension to be displayed in addition to the breakdown for each dimension value. To enable rollup for a dimension, call the rollup method on the dimension reference. Rollup is available for all dimension types.

```
dataset.data \
    ...
   .dimension( dataset.fields.date(hourly).rollup() ) \
   .dimension( dataset.fields.device.rollup() )
```

1.4.4 Filtering the query

A query can be filtered using several different filters. TODO TODO TODO TODO A metric filter is synonomous with the `HAVING` clause in a SQL query whereas a dimension filter corresponds to the `WHERE` clause. Dimension filters can also be applied to the display definition of Unique Dimensions.

When more than one filter is applied to a query, the results will be filtered to all rows/groups matching **all** of the conditions like a boolean `AND`. Some filters accept multiple values which create multiple conditions and filter data to rows/groups matching **any** of the conditions like a boolean `OR`.

Comparator (Metrics)

Comparator filters are created using standard operators:

- ==
- *!=*
- >
- >=
- <
- <=

```
dataset.data \
    ...
   .filter( dataset.fields.clicks >= 100 ) \
   .filter( dataset.fields.conversions == 1 )
```

Boolean (Boolean Dimensions)

Boolean filters only apply to boolean dimensions and filter whether the value of that boolean dimension is *True* or False‘ using the *.is_(True/False)* method on a `fireant.dataset.dimensions.BooleanDimension`.

```
dataset.data \
    ...
   .filter( dataset.fields.is_member.is_(True) )
```

Range (Date/Time dimensions)

Range filters apply to `fireant.dataset.dimensions.DatetimeDimension` dimensions using the *.between(start, end)* method. This is equivalent to a *BETWEEN* expression in the SQL query.

```
dataset.data \
    ...
   .filter( dataset.fields.date.between( datetime(2018, 8, 21), datetime(2019, 8, 20)␣
↪) )
```

Includes (Category and Unique dimensions)

Includes filters apply to `fireant.dataset.dimensions.CategoricalDimension` and `fireant.dataset.dimensions.UniqueDimension` dimensions using the *.isin(list)* method. Results will be included if they are equal to any of the values in the argument supplied.

Combining multiple include filters makes it possible to use both `AND` and `OR` filter logic.

```
dataset.data \
    ...
   .filter( dataset.fields.accounts.isin([1, 2, 3]) )
```

Excludes (Category and Unique dimensions)

Excludes filters are equivalent to Includes filters with negative logic. The same conditions apply using the *.notin(list)* method.

```
dataset.data \
    ...
   .filter( dataset.fields.accounts.notin([1, 2, 3]) )
```

Pattern (Category and Unique dimensions)

Pattern filters apply to `fireant.dataset.dimensions.CategoricalDimension` and `fireant.dataset.dimensions.UniqueDimension` dimensions using the *.like(*patterns)* method. They are the equivalent of a SQL `ILIKE` expression. The method accepts one or more pattern arguments which should be formatted for

SQL LIKE https://www.w3schools.com/sql/sql_like.asp. With multiple arguments, results are returned that match **any** of the patterns.

Combining multiple pattern filters makes it possible to use both `AND` and `OR` filter logic.

```
dataset.data \
    ...
   .filter( dataset.fields.device.like('desk%', 'mob%') )
```

Anti-Pattern

Anti-Pattern filters are equivalent to Pattern filters with negative logic. The same conditions apply using the *.not_like(*patterns)* method.

```
dataset = dataset(
    ...
    dimensions=[
        UniqueDimension('customer',
                        definition=customers.id,
                        display_definition=fn.Concat(customers.fname, ' ', customers.
↪lname))
    ],
    ...
)

dataset.data \
    ...
   .filter( dataset.fields.device.not_like('desk%', 'mob%') )
```

Filtering on Display Definitions

When using a `fireant.dataset.dimensions.UniqueDimension` with the `display_defintion` attribute, it is also possible to filter based on display values instead of the definition.

The `display` attribute on an instance of `fireant.dataset.dimensions.UniqueDimension` returns a `fireant.dataset.dimensions.DisplayDimension` which works like a normal dataset dimension. It works with the following filters: ‘

1.4.5 Visualizing the data with Widgets

At least one widget must be added to every query before calling the *fetch()* builder chain method. Each dataset query can return multiple widgets of different types, but because a dataset query resolves to a single SQL query, other parts of the query must be shared across all widgets, such as filters and dimensions.

Metrics are selected for each widget. Widgets can use the same metrics or different metrics in a query. The instance API for each widget is different since each widget uses metrics in different ways.

```
dataset.data \
    ...
   .widget( ... )
```

matplotlib

Coming soon!

pandas

The Pandas widget will return a Pandas data frame which is useful when displaying results in a Jupyter notebook. The Pandas widget is used by instantiating a *fireant.widgets.pandas.Pandas* class and passing one or more instances of *fireant.Field* as arguments.

The data frame will be structured with an index level for each dimension.

The Pandas widget takes additional arguments.

pivot [list[dimension]] A list of dimensions which should be pivoted as columns. If all dimensions are pivoted, the result will be identical to setting the `transpose` argument to `True`.

transpose [bool] When `True`, the data frame will be transposed.

sort [list[int]] A list of column indices to sort by. This sorts the data frame after it's been pivoted and transposed. Which columns are present depends on the selected dimensions and metrics as well as the `pivot` and `transponse` arguments.

```
from fireant import Pandas

Pandas( *metrics )
```

```
dataset.data \
    ...
   .dimension( dataset.dimension.date, dataset.dimension.device )
   .widget( Pandas(dataset.fields.clicks, dataset.fields.cost, dataset.fields.revenue,
                   pivot=(dataset.dimension.device, )
                   transpose=True) )
```

HighCharts

A HighCharts widget transforms the results into a HighCharts JSON config object. The widget is used by instantiating *fireant.widgets.highcharts.HighCharts* and calling the `axis` method with instances of *fireant.widgets.highcharts.Series* arguments. The axis method can be chained to create multiple axes.

Each *fireant.widgets.highcharts.Series* instance is constructed with one or more metrics.

```
from fireant import HighCharts

HighCharts( title ) \
    .axis ( HighCharts.LineChart( *metrics ), HighCharts.LineChart( *metrics ), ... )␣
↪\
    .axis ( HighCharts.BarChart( *metrics ) )
    ...
```

Datatables

React-Table

The React Table widget's instance API is identical to the Pandas widget, although it transforms results into a JSON config object meant for React-Table. See the section above on pandas for more information on the instance API.

```
from fireant import ReactTable
```

(continues on next page)

(continued from previous page)

```
dataset.data \
    ...
   .dimension( dataset.dimension.date, dataset.dimension.device )
   .widget( ReactTable(dataset.fields.clicks, dataset.fields.cost, dataset.fields.
↪revenue,
                       pivot=(dataset.dimension.device, )
                       transpose=True) )
```

1.4.6 Comparing Data to Previous Values using References

In some cases it is useful to compare the selected metrics over a period time such as in a Week-over-Week report. A *Reference* can be used to achieve this. *Reference* is a built-in function which can be chosen from the subclasses of *fireant.dataset.references.Reference*.

A *Reference* can be used as a fixed comparison, a change in value (delta), or a change in value as a percentage.

The *Reference* compares the currently selected data with itself shifted by the amount of the *Reference*.

The following options are available

- Day Over Day - Shifts by 1 day.
- Week Over Week - Shifts by 1 week.
- Month over Month - Shifts by 1 month.
- Quarter over Quarter - Shifts by 1 quarter or 3 months depending on whether the database backend supports quarter intervals.
- Year over Year - Shifts by 1 year.

For each *Reference*, there are the following variations:

- Delta - Difference in value
- Delta Percentage - Difference in value as a percentage of the previous value

A Date/Time dimension is required.

```
from fireant.dataset.references import WeekOverWeek

# Use a Week-over-Week reference
dataset.data \
    ...
   .reference( WeekOverWeek(dataset.fields.date) )

# Compare Week-over-Week change (delta)
dataset.data \
    ...
   .reference( WeekOverWeek(dataset.fields.date, delta=True) )

# Compare Week-over-Week change as a percentage (delta percentage)
dataset.data \
    ...
   .reference( WeekOverWeek(dataset.fields.date, delta=True, percent=True) )
```

Note: For any reference, the comparison is made for the same days of the week.

1.4.7 Post-Processing Operations

Operations include extra computations applied in python to the result of the SQL query to modify the result.

More on this later!

1.5 Extending *fireant*

1.6 API Reference

1.6.1 fireant package

Subpackages

fireant.database package

Submodules

fireant.database.base module

class fireant.database.base.**Database**(*host=None*, *port=None*, *database=None*, *max_processes=1*, *max_result_set_size=200000*, *cache_middleware=None*, *concurrency_middleware=None*)

Bases: object

This is a abstract base class used for interfacing with a database platform.

connect()
This function must establish a connection to the database platform and return it.

date_add(*field: pypika.terms.Term*, *date_part: str*, *interval: int*)
This function must add/subtract a Date or Date/Time object.

fetch(*query*)

get_column_definitions(*schema*, *table*)
This function must return the columns of a given schema and table.

query_cls
alias of pypika.queries.Query

slow_query_log_min_seconds = 15

to_char(*definition*)

trunc_date(*field*, *interval*)
This function must create a Pypika function which truncates a Date or DateTime object to a specific interval.

fireant.database.mysql module

class fireant.database.mysql.**DateAdd**(*field*, *interval_term*, *alias=None*)

Bases: pypika.terms.Function

Override for the MySQL specific DateAdd function which expects an interval instead of the date part and interval unit e.g. DATE_ADD("date", INTERVAL 1 YEAR)

class fireant.database.mysql.**MySQLDatabase**(*host='localhost'*, *port=3306*, *database=None*, *user=None*, *password=None*, *charset='utf8mb4'*, ***kwags*)

Bases: *fireant.database.base.Database*

MySQL client that uses the PyMySQL module.

connect()

Returns a MySQL connection

Returns pymysql Connection class

date_add(*field*, *date_part*, *interval*)

This function must add/subtract a Date or Date/Time object.

get_column_definitions(*schema*, *table*)

Return a list of column name, column data type pairs

query_cls

alias of pypika.dialects.MySQLQuery

to_char(*definition*)

trunc_date(*field*, *interval*)

This function must create a Pypika function which truncates a Date or DateTime object to a specific interval.

class fireant.database.mysql.**Trunc**(*field*, *date_format*, *alias=None*)

Bases: pypika.terms.Function

Wrapper for a custom MySQL TRUNC function (installed via a custom FireAnt MySQL script)

fireant.database.postgresql module

class fireant.database.postgresql.**DateTrunc**(*field*, *date_format*, *alias=None*)

Bases: pypika.terms.Function

Wrapper for the PostgreSQL date_trunc function

class fireant.database.postgresql.**PostgreSQLDatabase**(*host='localhost'*, *port=5432*, *database=None*, *user=None*, *password=None*, ***kwags*)

Bases: *fireant.database.base.Database*

PostgreSQL client that uses the psycopg module.

connect()

This function must establish a connection to the database platform and return it.

date_add(*field*, *date_part*, *interval*)

This function must add/subtract a Date or Date/Time object.

get_column_definitions(*schema*, *table*)

Return a list of column name, column data type pairs

query_cls

alias of pypika.dialects.PostgreSQLQuery

trunc_date(*field*, *interval*)
This function must create a Pypika function which truncates a Date or DateTime object to a specific interval.

fireant.database.redshift module

class fireant.database.redshift.**RedshiftDatabase**(*host='localhost'*, *port=5439*, *database=None*, *user=None*, *password=None*, ***kwags*)
Bases: *fireant.database.postgresql.PostgreSQLDatabase*

Redshift client that uses the psycopg module.

query_cls
alias of pypika.dialects.RedshiftQuery

fireant.database.snowflake module

class fireant.database.snowflake.**SnowflakeDatabase**(*user='snowflake'*, *password=None*, *account='snowflake'*, *database='snowflake'*, *private_key_data=None*, *private_key_password=None*, *region=None*, *warehouse=None*, ***kwags*)
Bases: *fireant.database.base.Database*

Snowflake client.

DATETIME_INTERVALS = {'day': 'DD', 'hour': 'HH', 'month': 'MM', 'quarter': 'Q', 'w

connect()
This function must establish a connection to the database platform and return it.

date_add(*field*, *date_part*, *interval*)
This function must add/subtract a Date or Date/Time object.

get_column_definitions(*schema*, *table*)
Return a list of column name, column data type pairs

query_cls
alias of pypika.dialects.SnowflakeQuery

trunc_date(*field*, *interval*)
This function must create a Pypika function which truncates a Date or DateTime object to a specific interval.

class fireant.database.snowflake.**Trunc**(*field*, *date_format*, *alias=None*)
Bases: pypika.terms.Function

Wrapper for TRUNC function for truncating dates.

fireant.database.vertica module

class fireant.database.vertica.**Trunc**(*field*, *date_format*, *alias=None*)
Bases: pypika.terms.Function

Wrapper for Vertica TRUNC function for truncating dates.

class fireant.database.vertica.**VerticaDatabase**(*host='localhost'*, *port=5433*, *database='vertica'*, *user='vertica'*, *password=None*, *read_timeout=None*, ***kwags*)

Bases: *fireant.database.base.Database*

Vertica client that uses the vertica_python driver.

DATETIME_INTERVALS = {'day': 'DD', 'hour': 'HH', 'month': 'MM', 'quarter': 'Q', 'w

connect()
This function must establish a connection to the database platform and return it.

date_add(*field*, *date_part*, *interval*)
This function must add/subtract a Date or Date/Time object.

get_column_definitions(*schema*, *table*)
Return schema information including column names and their data type

Parameters

- **schema** – the name of the schema if you would like to narrow the results down (String)
- **table** – the name of the table if you would like to narrow the results down (String)

Returns List of column name, column data type pairs

query_cls
alias of pypika.dialects.VerticaQuery

trunc_date(*field*, *interval*)
This function must create a Pypika function which truncates a Date or DateTime object to a specific interval.

fireant.dataset package

Submodules

fireant.dataset.fields module

exception fireant.dataset.fields.**DataSetFilterException**(*msg*, *allowed*)
Bases: Exception

class fireant.dataset.fields.**DataType**
Bases: enum.Enum

An enumeration.

boolean = 4

date = 1

number = 3

text = 2

class fireant.dataset.fields.**Field**(*alias*, *definition*, *data_type: fireant.dataset.fields.DataType = number*, *label=None*, *hint_table=None*, *prefix: str = None*, *suffix: str = None*, *thousands: str = None*, *precision: int = None*, *hyperlink_template: str = None*)

Bases: object

The *Field* class represents a field in the *DataSet* object. A field is a mapping to a column in a database query.

Parameters

- **alias** – A unique identifier used to identify the metric when writing slicer queries. This value must be unique over the metrics in the slicer.
- **definition** – A pypika expression which is used to select the value when building SQL queries. For metrics, this query **must** be aggregated, since queries always use a GROUP BY clause an metrics are not used as a group.
- **data_type** – {Number, Text, Boolean, Date} When True, the field's definition should be treated as an aggregate expression.
- **label** – (optional) A display value used for the metric. This is used for rendering the labels within the visualizations. If not set, the alias will be used as the default.
- **hint_table** – (optional) An optional table for fetching field choices. If this table is not set, the base table from the field definition will be used.
- **is_aggregate** – (default: False) When True, the field's definition should be treated as an aggregate expression.
- **precision** – (optional) A precision value for rounding decimals. By default, no rounding will be applied.
- **prefix** – (optional) A prefix for rendering labels in visualizations such as '$'
- **suffix** – A suffix for rendering labels in visualizations such as '€'

between(*lower*, *upper*)

Creates a filter to filter a slicer query.

Parameters

- **lower** – The start time of the filter. This is the beginning of the window for which results should be included.
- **upper** – The stop time of the filter. This is the end of the window for which results should be included.

Returns A slicer query filter used to filter a slicer query to results where this dimension is between the values start and stop.

eq(*other*)

ge(*other*)

gt(*other*)

is_(*value: bool*)

Creates a filter to filter a slicer query.

Parameters **value** – True or False

Returns A slicer query filter used to filter a slicer query to results where this dimension is True or False.

isin(*values: collections.abc.Iterable*)

Creates a filter that filters to rows where

Parameters **values** – An iterable of value to constrain the slicer query results by.

Returns A slicer query filter used to filter a slicer query to results where this dimension is one of a set of values. Opposite of #notin.

le(*other*)

like(*pattern*, **patterns*)

lt(*other*)

ne(*other*)

not_like(*pattern*, **patterns*)

notin(*values*)
Creates a filter to filter a slicer query.

Parameters **values** – An iterable of value to constrain the slicer query results by.

Returns A slicer query filter used to filter a slicer query to results where this dimension is *not* one of a set of values. Opposite of #isin.

share

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

void()

fireant.dataset.fields.**restrict_types**(*allowed*)

fireant.dataset.filters module

class fireant.dataset.filters.**AntiPatternFilter**(*field_alias*, *dimension_definition*, *pattern*, **patterns*)
Bases: *fireant.dataset.filters.PatternFilter*

class fireant.dataset.filters.**BooleanFilter**(*field_alias*, *dimension_definition*, *value*)
Bases: *fireant.dataset.filters.Filter*

class fireant.dataset.filters.**ComparatorFilter**(*field_alias*, *metric_definition*, *operator*, *value*)
Bases: *fireant.dataset.filters.Filter*

class Operator
Bases: object

eq = 'eq'

gt = 'gt'

gte = 'gte'

lt = 'lt'

lte = 'lte'

ne = 'ne'

class fireant.dataset.filters.**ContainsFilter**(*field_alias*, *dimension_definition*, *values*)
Bases: *fireant.dataset.filters.Filter*

class fireant.dataset.filters.**ExcludesFilter**(*field_alias*, *dimension_definition*, *values*)
Bases: *fireant.dataset.filters.Filter*

class fireant.dataset.filters.**Filter**(*field_alias*, *definition*)
Bases: object

is_aggregate

class fireant.dataset.filters.**PatternFilter**(*field_alias*, *dimension_definition*, *pattern*, **patterns*)
Bases: *fireant.dataset.filters.Filter*

class fireant.dataset.filters.**RangeFilter**(*field_alias*, *dimension_definition*, *start*, *stop*)
Bases: *fireant.dataset.filters.Filter*

class fireant.dataset.filters.**VoidFilter**(*field_alias*)
Bases: *fireant.dataset.filters.Filter*

fireant.dataset.intervals module

class fireant.dataset.intervals.**DatetimeInterval**(*dimension*, *interval_key*)
Bases: *fireant.dataset.modifiers.DimensionModifier*

class fireant.dataset.intervals.**NumericInterval**(*dimension*, *size=1*, *offset=0*)
Bases: *fireant.dataset.modifiers.DimensionModifier*

fireant.dataset.joins module

class fireant.dataset.joins.**Join**(*table*, *criterion*, *join_type=<JoinType.inner: ">*)
Bases: object

WRITEME

fireant.dataset.klass module

class fireant.dataset.klass.**DataSet**(*table*, *database*, *joins=()*, *fields=()*, *always_query_all_metrics=False*)
Bases: object

WRITEME

class Fields(*items*)
Bases: fireant.dataset.klass._Container

fireant.dataset.modifiers module

class fireant.dataset.modifiers.**DimensionModifier**(*wrapped*)
Bases: *fireant.dataset.modifiers.Modifier*

wrapped_key = 'dimension'

class fireant.dataset.modifiers.**FilterModifier**(*wrapped*)
Bases: *fireant.dataset.modifiers.Modifier*

wrapped_key = 'filter'

class fireant.dataset.modifiers.**Modifier**(*wrapped*)
Bases: object

wrapped_key = 'wrapped'

class fireant.dataset.modifiers.**OmitFromRollup**(*wrapped*)
Bases: *fireant.dataset.modifiers.FilterModifier*

class fireant.dataset.modifiers.**Rollup**(*wrapped*)
Bases: *fireant.dataset.modifiers.DimensionModifier*

definition

fireant.dataset.operations module

class fireant.dataset.operations.**CumMean**(*arg*)
Bases: fireant.dataset.operations._Cumulative

apply(*data_frame*, *reference*)

static cummean(*x*)

class fireant.dataset.operations.**CumProd**(*arg*)
Bases: fireant.dataset.operations._Cumulative

apply(*data_frame*, *reference*)

class fireant.dataset.operations.**CumSum**(*arg*)
Bases: fireant.dataset.operations._Cumulative

apply(*data_frame*, *reference*)

class fireant.dataset.operations.**Operation**
Bases: object

The *Operation* class represents an operation in the *Slicer* API.

apply(*data_frame*, *reference*)

metrics

operations

class fireant.dataset.operations.**RollingMean**(*arg*, *window*, *min_periods=None*)
Bases: *fireant.dataset.operations.RollingOperation*

apply(*data_frame*, *reference*)

rolling_mean(*x*)

class fireant.dataset.operations.**RollingOperation**(*arg*, *window*, *min_periods=None*)
Bases: fireant.dataset.operations._BaseOperation

apply(*data_frame*, *reference*)

class fireant.dataset.operations.**Share**(*metric: fireant.dataset.fields.Field*, *over: fireant.dataset.fields.Field = None*, *precision=2*)
Bases: fireant.dataset.operations._BaseOperation

apply(*data_frame*, *reference*)

metric

metrics

over

fireant.dataset.references module

class fireant.dataset.references.**Reference**(*field*, *reference_type*, *delta=False*, *delta_percent=False*)

Bases: object

class fireant.dataset.references.**ReferenceType**(*alias*, *label*, *time_unit: str*, *interval: int*)

Bases: object

fireant.dataset.references.**reference_term**(*reference: fireant.dataset.references.Reference*, *original_query: pypika.queries.QueryBuilder*, *ref_query: pypika.queries.QueryBuilder*)

Part of query building. Given a reference, the original slicer query, and the ref query, creates the pypika for the reference that should be selected in the reference container query.

Parameters

- **reference** –
- **original_query** –
- **ref_query** –

Returns

fireant.dataset.totals module

fireant.dataset.totals.**get_totals_marker_for_dtype**(*dtype*)

For a given dtype, return the index value to use to indicate that the data frame row contains totals.

Parameters **dtype** –

Returns

fireant.dataset.totals.**scrub_totals_from_share_results**(*data_frame*, *dimensions*)

This function returns a data frame with the values for dimension totals filtered out if the corresponding dimension was not queried with rollup. This comes into play when the share operation is used on metrics which requires the totals across the values.

There are two different logical branches for this function that perform the same work, one on a single-level index data frame and another for a multi-level index.

Parameters

- **data_frame** – The result data set.
- **dimensions** – A list of dimensions that were queried for to produce the result data set.

Returns The data frame with totals rows removed for dimensions that were not queried with rollup.

fireant.middleware package

Submodules

fireant.middleware.concurrency module

class fireant.middleware.concurrency.**BaseConcurrencyMiddleware**

Bases: abc.ABC

The abstract base class that should be inherited from to define a concurrency middleware.

fetch_queries_as_dataframe(*queries*, *database*)
Implementations of this method should execute the given queries on the supplied database and return the results. :return: A list of the results of the executed queries.

fetch_query(*query*, *database*)
Perform a query on the database.

Parameters

- **query** – The query to execute.
- **database** – The database to perform the query on.

Returns The result of the query.

class fireant.middleware.concurrency.**ThreadPoolConcurrencyMiddleware**(*max_processes=1*)
Bases: *fireant.middleware.concurrency.BaseConcurrencyMiddleware*

A concurrency middleware implementation based on threadpools used as a default middleware.

fetch_queries_as_dataframe(*queries*, *database*)
Executes the different queries in separate threads.

fireant.queries package

Submodules

fireant.queries.builder module

class fireant.queries.builder.**DataSetQueryBuilder**(*dataset*)
Bases: *fireant.queries.builder.QueryBuilder*

Slicer queries consist of widgets, dimensions, filters, and references. At least one or more widgets is required. All others are optional.

dimension(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

fetch(*hint=None*) → Iterable[Dict]
Fetch the data for this query and transform it into the widgets.

Parameters **hint** – A query hint label used with database vendors which support it. Adds a label comment to the query.

Returns A list of dict (JSON) objects containing the widget configurations.

orderby(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

plot()

reference(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

reference_groups

sql
Serialize this query builder to a list of Pypika/SQL queries. This function will return one query for every combination of reference and rolled up dimension (including null options).

This collects all of the metrics in each widget, dimensions, and filters and builds a corresponding pypika query to fetch the data. When references are used, the base query normally produced is wrapped in an outer query and a query for each reference is joined based on the referenced dimension shifted.

widget (**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

class fireant.queries.builder.**DimensionChoicesQueryBuilder** (*dataset*, *dimension*)
Bases: *fireant.queries.builder.QueryBuilder*

This builder is used for building slicer queries for fetching the choices for a dimension given a set of filters.

fetch (*hint=None*, *force_include=()*) → pandas.core.series.Series
Fetch the data for this query and transform it into the widgets.

Parameters

- **hint** – For database vendors that support it, add a query hint to collect analytics on the queries triggered by fireant.
- **force_include** – A list of dimension values to include in the result set. This can be used to avoid having necessary results cut off due to the pagination. These results will be returned at the head of the results.

Returns A list of dict (JSON) objects containing the widget configurations.

sql
Serializes this query builder as a set of SQL queries. This method will always return a list of one query since only one query is required to retrieve dimension choices.

The slicer query extends this with metrics, references, and totals.

class fireant.queries.builder.**DimensionLatestQueryBuilder** (*dataset*)
Bases: *fireant.queries.builder.QueryBuilder*

fetch (*hint=None*)
Fetches the data for this query instance and returns it in an instance of *pd.DataFrame*

Parameters **hint** – For database vendors that support it, add a query hint to collect analytics on the queries triggerd by fireant.

sql
Serializes this query builder as a set of SQL queries. This method will always return a list of one query since only one query is required to retrieve dimension choices.

This function only handles dimensions (select+group by) and filtering (where/having), which is everything needed for the query to fetch choices for dimensions.

The slicer query extends this with metrics, references, and totals.

class fireant.queries.builder.**QueryBuilder** (*dataset*, *table*)
Bases: object

This is the base class for building slicer queries. This class provides an interface for building slicer queries via a set of functions which can be chained together.

fetch (*hint=None*)
Fetches the data for this query instance and returns it in an instance of *pd.DataFrame*

Parameters **hint** – For database vendors that support it, add a query hint to collect analytics on the queries triggerd by fireant.

filter(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

limit(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

offset(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

sql

Serialize this query builder object to a set of Pypika/SQL queries.

This is the base implementation shared by two implementations: the query to fetch data for a slicer request and the query to fetch choices for dimensions.

This function only handles dimensions (select+group by) and filtering (where/having), which is everything needed for the query to fetch choices for dimensions.

The slicer query extends this with metrics, references, and totals.

exception fireant.queries.builder.**QueryException**

Bases: *fireant.exceptions.SlicerException*

fireant.queries.builder.**add_hints**(*queries*, *hint=None*)

fireant.queries.builder.**get_column_names**(*database*, *table*)

fireant.queries.execution module

fireant.queries.execution.**db_cache**(*func*)

fireant.queries.execution.**fetch_as_dataframe**(*query: str*, *database: fireant.database.base.Database*)

Executes a query to fetch data from database middleware and builds/cleans the data as a data frame. The query execution is logged with its duration.

Parameters

- **database** – instance of *fireant.Database*, database middleware
- **query** – Query string

Returns *pd.DataFrame* constructed from the result of the query

fireant.queries.execution.**fetch_data**(*database: fireant.database.base.Database*, *queries: Union[collections.abc.Sized, Iterable]*, *dimensions: Iterable[fireant.dataset.fields.Field]*, *share_dimensions: Iterable[fireant.dataset.fields.Field] = ()*, *reference_groups=()*)

fireant.queries.execution.**log**(*func*)

fireant.queries.execution.**reduce_result_set**(*results: Iterable[pandas.core.frame.DataFrame]*, *reference_groups*, *dimensions: Iterable[fireant.dataset.fields.Field]*, *share_dimensions: Iterable[fireant.dataset.fields.Field]*)

Reduces the result sets from individual queries into a single data frame. This effectively joins sets of references and concatenates the sets of totals.

Parameters

- **results** – A list of data frame
- **reference_groups** – A list of groups of references (grouped by interval such as WoW, etc)
- **dimensions** – A list of dimensions, used for setting the index on the result data frame.
- **share_dimensions** – A list of dimensions from which the totals are used for calculating share operations.

Returns

fireant.queries.field_helper module

fireant.queries.field_helper.**make_orders_for_dimensions**(*dimensions*)

Creates a list of ordering for a slicer query based on a list of dimensions. The dimensions's display definition is used preferably as the ordering term but the definition is used for dimensions that do not have a display definition.

Parameters **dimensions** –

Returns a list of tuple pairs like (term, orientation) for ordering a SQL query where the first element is the term to order by and the second is the orientation of the ordering, ASC or DESC.

fireant.queries.field_helper.**make_term_for_dimension**(*dimension*, *window=None*)

Makes a list of pypika terms for a given slicer definition.

Parameters

- **dimension** – A slicer dimension.
- **window** – A window function to apply to the dimension definition if it is a continuous dimension.

Returns a list of terms required to select and group by in a SQL query given a slicer dimension. This list will contain either one or two elements. A second element will be included if the dimension has a definition for its display field.

fireant.queries.field_helper.**make_term_for_metrics**(*metric*)

fireant.queries.finders module

exception fireant.queries.finders.**CircularJoinsException**

Bases: *fireant.exceptions.SlicerException*

exception fireant.queries.finders.**MissingTableJoinException**

Bases: *fireant.exceptions.SlicerException*

class fireant.queries.finders.**ReferenceGroup**(*dimension*, *time_unit*, *intervals*)

Bases: tuple

dimension

Alias for field number 0

intervals

Alias for field number 2

time_unit
Alias for field number 1

fireant.queries.finders.**find_and_group_references_for_dimensions**(*dimensions*, *references*)
Finds all of the references for dimensions and groups them by dimension, interval unit, number of intervals.

This structure reflects how the references need to be joined to the slicer query. References of the same type (WoW, WoW.delta, WoW.delta_percent) can share a join query.

Parameters

- **dimensions** –
- **references** –

Returns

An *OrderedDict* where the keys are 3-item tuples consisting of "Dimension, interval unit, # of intervals.

```
Example
{
    (Dimension(date_1), 'weeks', 1): [WoW, WoW.delta],
    (Dimension(date_1), 'years', 1): [YoY],
    (Dimension(date_7), 'days', 1): [DoD, DoD.delta_percent],
}
```

fireant.queries.finders.**find_and_replace_reference_dimensions**(*references*, *dimensions*)
Finds the dimension for a reference in the query if there is one and replaces it. This is to force the reference to use the same modifiers with a dimension if it is selected in the query.

Parameters

- **references** –
- **dimensions** –

Returns

fireant.queries.finders.**find_filters_for_totals**(*filters*)

Parameters **filters** –

Returns a list of filters that should be applied to totals queries. This removes any filters from the list that have the *OmitFromRollup* modifier applied to them.

fireant.queries.finders.**find_joins_for_tables**(*joins*, *base_table*, *required_tables*)
Given a set of tables required for a slicer query, this function finds the joins required for the query and sorts them topologically.

Returns A list of joins in the order that they must be joined to the query.

Raises MissingTableJoinException - If a table is required but there is no join for that table CircularJoinsException - If there is a circular dependency between two or more joins

fireant.queries.finders.**find_metrics_for_widgets**(*widgets*)

Returns an ordered, distinct list of metrics used in all widgets as part of this query.

fireant.queries.finders.**find_operations_for_widgets**(*widgets*)

Returns an ordered, distinct list of metrics used in all widgets as part of this query.

fireant.queries.finders.**find_required_tables_to_join**(*elements*, *base_table*)
Collect all the tables required for a given list of slicer elements. This looks through the definition and display_definition attributes of all elements and

This looks through the metrics, dimensions, and filter included in this slicer query. It also checks both the definition field of each element as well as the display definition for Unique Dimensions.

Returns A collection of tables required to execute a query,

fireant.queries.finders.**find_share_dimensions**(*dimensions*, *operations*)
Returns a subset list of dimensions from the list of dimensions that are used as the over-dimension in share operations.

Parameters

- **dimensions** –
- **operations** –

Returns

fireant.queries.finders.**find_totals_dimensions**(*dimensions*, *share_dimensions*)

Parameters

- **dimensions** –
- **share_dimensions** –

Returns an list of all dimension field in the list argument *dimensions* which have the *Rollup* modifier applied to them or are used as a basis for a share metric.

fireant.queries.pagination module

fireant.queries.pagination.**paginate**(*data_frame*, *widgets*, *orders=()*, *limit=None*, *offset=None*)

Parameters

- **data_frame** – The result set to paginate.
- **widgets** – An iterable of widgets that the pagination is being applied for.
- **orders** – An iterable of (<Dimension/Metric>, pypika.Order)
- **limit** – A limit of the number of data points/series
- **offset** – A offset of the number of data points/series

Returns A paginated data frame. If the widget required grouped pagination, then there should be an upperbound *limit*(n_index_level_0)*. Otherwise the data frame should have the same length as the limit.

fireant.queries.reference_helper module

fireant.queries.reference_helper.**adapt_for_reference_query**(*reference_parts*, *database*, *dimensions*, *metrics*, *filters*, *references*)

fireant.queries.slow_query_logger module

fireant.queries.special_cases module

fireant.queries.special_cases.**adjust_dataframe_for_rolling_window**(*operations*, *data_frame*)

This function adjusts the resulting data frame after executing a slicer query with a rolling operation. If there is a date dimension in the first level of the data frame's index and a rolling operation is applied, it will slice the dates following the max window to remove it. This way, the adjustment of date filters applied in #adjust_daterange_filter_for_rolling_window are removed from the data frame but also in case there are no filters, the first few date data points will be removed where the rolling window cannot be calculated.

Parameters

- **operations** –
- **data_frame** –

Returns

fireant.queries.special_cases.**adjust_daterange_filter_for_rolling_window**(*dimensions*, *operations*, *filters*)

This function adjusts date filters for a rolling operation in order to select enough date to compute the values for within the original range.

It only applies when using a date dimension in the first position and a RangeFilter is used on that dimension. It is meant to be applied to a slicer query.

Parameters

- **dimensions** – The dimensions applied to a slicer query
- **operations** – The dimensions used in widgets in a slicer query
- **filters** – The filters applied to a slicer query

Returns

fireant.queries.special_cases.**apply_operations_to_data_frame**(*operations*, *data_frame*)

fireant.queries.special_cases.**apply_special_cases**(*f*)

fireant.queries.special_cases.**apply_to_query_args**(*database*, *table*, *joins*, *dimensions*, *metrics*, *operations*, *filters*, *references*, *orders*)

fireant.queries.sql_transformer module

fireant.queries.sql_transformer.**make_latest_query**(*database: fireant.database.base.Database*, *base_table: pypika.queries.Table*, *joins: Iterable[fireant.dataset.joins.Join] = ()*, *dimensions: Iterable[fireant.dataset.fields.Field] = ()*)

fireant.queries.sql_transformer.**make_slicer_query**(*database: fireant.database.base.Database*, *base_table: pypika.queries.Table*, *joins: Iterable[fireant.dataset.joins.Join] = ()*, *dimensions: Iterable[fireant.dataset.fields.Field] = ()*, *metrics: Iterable[fireant.dataset.fields.Field] = ()*, *filters: Iterable[fireant.dataset.filters.Filter] = ()*, *orders: Iterable = ()*)

Creates a pypika/SQL query from a list of slicer elements.

This is the base implementation shared by two implementations: the query to fetch data for a slicer request and the query to fetch choices for dimensions.

This function only handles dimensions (select+group by) and filtering (where/having), which is everything needed for the query to fetch choices for dimensions.

The slicer query extends this with metrics, references, and totals.

Parameters

- **database** –
- **base_table** – pypika.Table - The base table of the query, the one in the FROM clause
- **joins** – A collection of joins available in the slicer. This should include all slicer joins. Only joins required for the query will be used.
- **dimensions** – A collection of dimensions to use in the query.
- **metrics** – A collection of metrics to use in the query.
- **filters** – A collection of filters to apply to the query.
- **orders** – A collection of orders as tuples of the metric/dimension to order by and the direction to order in.

Returns

fireant.queries.sql_transformer.**make_slicer_query_with_totals_and_references**(*database*, *table*, *joins*, *dimensions*, *metrics*, *operations*, *filters*, *references*, *orders*, *share_dimensions=(*

Parameters

- **database** –
- **table** –
- **joins** –
- **dimensions** –
- **metrics** –
- **operations** –
- **filters** –
- **references** –
- **orders** –
- **share_dimensions** –

Returns

fireant.queries.totals_helper module

fireant.queries.totals_helper.**adapt_for_totals_query**(*totals_dimension*, *dimensions*, *filters*)

Adapt filters for totals query. This function will select filters for total dimensions depending on the apply_filter_to_totals values for the filters. A total dimension with value None indicates the base query for which all filters will be applied by default.

Parameters

- **totals_dimension** –
- **dimensions** –
- **filters** –

Returns

fireant.widgets package

Submodules

fireant.widgets.base module

exception fireant.widgets.base.**MetricRequiredException**
Bases: *fireant.exceptions.SlicerException*

class fireant.widgets.base.**ReferenceItem**(*item*, *reference*)
Bases: object

class fireant.widgets.base.**TransformableWidget**(**items*)
Bases: *fireant.widgets.base.Widget*

group_pagination = False

transform(*data_frame*, *slicer*, *dimensions*, *references*)

- Main entry point -

Transformers the result set *pd.DataFrame* from a slicer query into the output format for this specific widget type.

Parameters

- **data_frame** – The data frame containing the data. Index must match the dimensions parameter.
- **slicer** – The slicer that is in use.
- **dimensions** – A list of dimensions that are being rendered.
- **references** – A list of references that are being rendered.

Returns A dict meant to be dumped as JSON.

class fireant.widgets.base.**Widget**(**items*)
Bases: object

item(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

metrics

operations

fireant.widgets.chart_base module

class fireant.widgets.chart_base.**Axis**(*series: Iterable[fireant.widgets.chart_base.Series]*, *label=None*, *y_axis_visible=True*)
Bases: object

class fireant.widgets.chart_base.**ChartWidget**(**items*)
Bases: *fireant.widgets.base.Widget*

class AreaPercentageSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: fireant.widgets.chart_base.AreaSeries

stacking = 'percent'

class AreaSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: *fireant.widgets.chart_base.ContinuousAxisSeries*

type = 'area'

class AreaStackedSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: fireant.widgets.chart_base.AreaSeries

stacking = 'normal'

class BarSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: *fireant.widgets.chart_base.Series*

type = 'bar'

class ColumnSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: *fireant.widgets.chart_base.Series*

type = 'column'

class LineSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: *fireant.widgets.chart_base.ContinuousAxisSeries*

type = 'line'

class PieSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: *fireant.widgets.chart_base.Series*

type = 'pie'

class StackedBarSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: fireant.widgets.chart_base.BarSeries

stacking = 'normal'

class StackedColumnSeries(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation], stacking=None*)

Bases: fireant.widgets.chart_base.ColumnSeries

stacking = 'normal'

axis(**args*, *mutate=False*, ***kwargs*)

Parameters **mutate** – When True, overrides the immutable behavior of this decorator.

metrics

Returns A set of metrics used in this chart. This collects all metrics across all axes.

operations

class fireant.widgets.chart_base.**ContinuousAxisSeries**(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation]*, *stacking=None*)

Bases: *fireant.widgets.chart_base.Series*

class fireant.widgets.chart_base.**Series**(*metric: Union[fireant.dataset.fields.Field, fireant.dataset.operations.Operation]*, *stacking=None*)

Bases: object

needs_marker = False

stacking = None

type = None

fireant.widgets.csv module

class fireant.widgets.csv.**CSV**(*metric: fireant.dataset.fields.Field*, **metrics*, *group_pagination=False*, ***kwargs*)

Bases: *fireant.widgets.pandas.Pandas*

transform(*data_frame*, *slicer*, *dimensions*, *references*)

WRITEME

Parameters

- **data_frame** –
- **slicer** –
- **dimensions** –
- **references** –

Returns

fireant.widgets.highcharts module

class fireant.widgets.highcharts.**HighCharts**(*title=None*, *colors=None*, *x_axis_visible=True*, *tooltip_visible=True*)

Bases: *fireant.widgets.chart_base.ChartWidget*, *fireant.widgets.base.TransformableWidget*

group_pagination = True

transform(*data_frame*, *slicer*, *dimensions*, *references*)

- Main entry point -

Transforms a data frame into HighCharts JSON format.

See https://api.highcharts.com/highcharts/

Parameters

- **data_frame** – The data frame containing the data. Index must match the dimensions parameter.
- **slicer** – The slicer that is in use.

- **dimensions** – A list of dimensions that are being rendered.
- **references** – A list of references that are being rendered.

Returns A dict meant to be dumped as JSON.

fireant.widgets.matplotlib module

class fireant.widgets.matplotlib.**Matplotlib**(*title=None*)

Bases: *fireant.widgets.chart_base.ChartWidget*, *fireant.widgets.base.TransformableWidget*

static get_plot_func_for_series_type(*pd_series*, *label*, *chart_series*)

transform(*data_frame*, *slicer*, *dimensions*, *references*)

- Main entry point -

Transformers the result set *pd.DataFrame* from a slicer query into the output format for this specific widget type.

Parameters

- **data_frame** – The data frame containing the data. Index must match the dimensions parameter.
- **slicer** – The slicer that is in use.
- **dimensions** – A list of dimensions that are being rendered.
- **references** – A list of references that are being rendered.

Returns A dict meant to be dumped as JSON.

fireant.widgets.pandas module

class fireant.widgets.pandas.**Pandas**(*metric: fireant.dataset.fields.Field*, **metrics*, *pivot=()*, *transpose=False*, *sort=None*, *ascending=None*, *max_columns=None*)

Bases: *fireant.widgets.base.TransformableWidget*

add_formatting(*dimensions*, *items*, *pivot_df*)

pivot_data_frame(*data_frame*, *pivot=()*, *transpose=False*)

Pivot and transpose the data frame. Dimensions including in the *pivot* arg will be unshifted to columns. If *transpose* is True the data frame will be transposed. If there is only index level in the data frame (ie. one dimension), and that dimension is pivoted, then the data frame will just be transposed. If there is a single metric in the data frame and at least one dimension pivoted, the metrics column level will be dropped for simplicity.

Parameters

- **data_frame** – The result set data frame
- **pivot** – A list of index aliases for *data_frame* of levels to shift
- **transpose** – A boolean true or false whether to transpose the data frame.

Returns The shifted/transposed data frame

sort_data_frame(*data_frame*)

transform(*data_frame*, *slicer*, *dimensions*, *references*)
WRITEME

Parameters

- **data_frame** –
- **slicer** –
- **dimensions** –
- **references** –

Returns

fireant.widgets.reacttable module

class fireant.widgets.reacttable.**ReactTable**(*metric*, **metrics*, *pivot=()*, *transpose=False*, *sort=None*, *ascending=None*, *max_columns=None*)

Bases: *fireant.widgets.pandas.Pandas*

This component does not work with react-table out of the box, some customization is needed in order to work with the transformed data.

```
// A Custom TdComponent implementation is required by Fireant in order to render␣
↪display values
const TdComponent = ({
                       toggleSort,
                       className,
                       children,
                       ...rest
                     }) =>
    <div className={classNames('rt-td', className)} role="gridcell" {...rest}>
        {_.get(children, 'display', children.raw) || <span> </span>}
    </div>;

const FireantReactTable = ({
                        config, // The payload from fireant
                      }) =>
    <ReactTable columns={config.columns}
                data={config.data}
                minRows={0}

                TdComponent={ DashmoreTdComponent}
                defaultSortMethod={(a, b, desc) => ReactTableDefaults.
↪defaultSortMethod(a.raw, b.raw, desc)}>
    </ReactTable>;
```

static format_data_frame(*data_frame*)
This function prepares the raw data frame for transformation by formatting dates in the index and removing any remaining NaN/NaT values. It also names the column as metrics so that it can be treated like a dimension level.

Parameters

- **data_frame** – The result set data frame
- **dimensions** –

Returns

static map_hyperlink_templates(*df*, *dimensions*)

Creates a mapping for each dimension to it's hyperlink template if it is possible to create the hyperlink template for it.

The hyperlink template is a URL-like string containing curley braces enclosing dimension keys: *{dimension}*. While rendering this widget, the dimension key placeholders need to be replaced with the dimension values for that row.

Parameters

- **df** – The result data set that is being transformed. The data frame SHOULD be pivoted/transposed if that step is required, before calling this function, in order to prevent the template from being included for the dimension if one of the required dimensions is pivoted.
- **dimensions** – The list of dimensions included in the query that created the result data set df.

Returns A dict with the dimension key as the key and the hyperlink template as the value. Templates will only be included if it will be possible to fill in the required parameters.

transform(*data_frame*, *slicer*, *dimensions*, *references*)

Transforms a data frame into a format for ReactTable. This is an object containing attributes *columns* and *data* which align with the props in ReactTable with the same name.

Parameters

- **data_frame** – The result set data frame
- **slicer** – The slicer that generated the data query
- **dimensions** – A list of dimensions that were selected in the data query
- **references** – A list of references that were selected in the data query

Returns An dict containing attributes *columns* and *data* which align with the props in ReactTable with the same names.

static transform_data(*data_frame*, *field_map*, *dimension_hyperlink_templates*, *is_transposed*)

Builds a list of dicts containing the data for ReactTable. This aligns with the accessors set by #transform_dimension_column_headers and #transform_metric_column_headers

Parameters

- **data_frame** – The result set data frame
- **field_map** – A mapping to all the fields in the slicer used for this query.
- **dimension_hyperlink_templates** –
- **is_transposed** –

static transform_data_column_headers(*data_frame*, *field_map*)

Convert the metrics into ReactTable column header definitions. This includes any pivoted dimensions, which will result in multiple rows of headers.

Returns

Parameters

- **data_frame** – The result set data frame
- **field_map** – A map to find metrics/operations based on their keys found in the data frame.

Returns A list of column header definitions with the following structure.

```
columns = [{
  Header: 'Column A',
  columns: [{
    Header: 'SubColumn A.0',
    accessor: 'a.0',
  }, {
    Header: 'SubColumn A.1',
    accessor: 'a.1',
  }]
}, {
  Header: 'Column B',
  columns: [
    ...
  ]
}]
```

static transform_index_column_headers(*data_frame*, *field_map*)

Convert the un-pivoted dimensions into ReactTable column header definitions.

Parameters

- **data_frame** – The result set data frame
- **field_map** –

Returns A list of column header definitions with the following structure.

```
columns = [{
  Header: 'Column A',
  accessor: 'a',
}, {
  Header: 'Column B',
  accessor: 'b',
}]
```

static transform_row_index(*index_values*, *field_map*, *dimension_hyperlink_templates*)

static transform_row_values(*series*, *fields*, *is_transposed*)

class fireant.widgets.reacttable.**TotalsItem**

Bases: object

alias = '$totals'

label = 'Totals'

precision = None

prefix = None

suffix = None

fireant.widgets.reacttable.**map_index_level**(*index*, *level*, *func*)

fireant.widgets.reacttable.**return_none**(**args*, ***kwargs*)

Submodules

fireant.exceptions module

exception fireant.exceptions.**SlicerException**
Bases: Exception

fireant.formats module

fireant.formats.**date_as_millis**(**args*, ***kwargs*)

fireant.formats.**date_as_string**(**args*, ***kwargs*)

fireant.formats.**display_value**(*value*, *field*, *date_as=functools.partial(<function apply_kwargs>, <function date_as_string>)*)
Converts a metric value into the display value by applying formatting.

Parameters

- **value** – The raw metric value.
- **field** – The slicer field that the value represents.
- **date_as** – A format function for datetimes.

Returns A formatted string containing the display value for the metric.

fireant.formats.**json_value**(*value*)
This function will return only values safe for JSON

fireant.formats.**raw_value**(*value*, *field*, *date_as=functools.partial(<function apply_kwargs>, <function date_as_string>)*)
Converts a raw metric value into a safe type. This will change dates into strings, NaNs into Nones, and np types into their corresponding python types.

Parameters

- **value** – The raw metric value.
- **field** – The slicer field that the value represents.
- **date_as** –

fireant.formats.**return_none**(**args*, ***kwargs*)

fireant.formats.**safe_value**(*value*)

fireant.formats.**wrap_styling**(**args*, ***kwargs*)

fireant.reference_helpers module

fireant.reference_helpers.**reference_alias**(*metric*, *reference*)
Format a metric key for a reference.

Returns A string that is used as the key for a reference metric.

fireant.reference_helpers.**reference_label**(*metric*, *reference*)
Format a metric label for a reference.

Returns A string that is used as the display value for a reference metric.

fireant.reference_helpers.**reference_prefix**(*metric*, *reference*)
Return the prefix for a metric displayed for a reference (or no Reference)

Returns A string that is used as the prefix for a reference metric.

fireant.reference_helpers.**reference_suffix**(*metric*, *reference*)
Return the suffix for a metric displayed for a reference (or no Reference)

Returns A string that is used as the suffix for a reference metric.

fireant.settings module

fireant.utils module

fireant.utils.**alias_for_alias_selector**(*f_alias*)

fireant.utils.**alias_selector**(*alias*)

fireant.utils.**apply_kwargs**(*f*, **args*, ***kwargs*)

fireant.utils.**chunks**(*l*, *n*)
Yield successive n-sized chunks from l.

fireant.utils.**filter_kwargs**(*f*)
Removes any kwargs from function call that are not accepted by the called function.

Parameters **f** –

Returns

fireant.utils.**flatten**(*items*)

fireant.utils.**getdeepattr**(*d*, *keys*, *default_value=None*)
Similar to the built-in *getattr*, this function accepts a list/tuple of keys to get a value deep in a *dict*

Given the following dict structure

```
d = {
  'A': {
    '0': {
      'a': 1,
      'b': 2,
    }
  },
}
```

Calling *getdeepattr* with a key path to a value deep in the structure will return that value. If the value or any of the objects in the key path do not exist, then the default value is returned.

```
assert 1 == getdeepattr(d, ('A', '0', 'a'))
assert 2 == getdeepattr(d, ('A', '0', 'b'))
assert 0 == getdeepattr(d, ('A', '0', 'c'), default_value=0)
assert 0 == getdeepattr(d, ('X', '0', 'a'), default_value=0)
```

Parameters

- **d** – A dict value with nested dict attributes.
- **keys** – A list/tuple path of keys in *d* to the desired value
- **default_value** – A default value that will be returned if the path *keys* does not yield a value.

Returns The value following the path *keys* or *default_value*

fireant.utils.**groupby**(*items*, *by*)
Group items using a function to derive a key.

Parameters

- **items** – The items to group
- **by** – A lambda function to create a key based on the item

Returns an Ordered dict

fireant.utils.**immutable**(*func*)
Decorator for wrapper "builder" functions. These are functions on the Query class or other classes used for building queries which mutate the query and return self. To make the build functions immutable, this decorator is used which will deepcopy the current instance. This decorator will return the return value of the inner function or the new copy of the instance. The inner function does not need to return self.

fireant.utils.**ordered_distinct_list**(*l*)

fireant.utils.**ordered_distinct_list_by_attr**(*l*, *attr='alias'*)

fireant.utils.**reduce_data_frame_levels**(*data_frame*, *level*)

fireant.utils.**setdeepattr**(*d*, *keys*, *value*)
Similar to the built-in *setattr*, this function accepts a list/tuple of keys to set a value deep in a *dict*

Given the following dict structure

```
d = {
  'A': {
    '0': {
      'a': 1,
      'b': 2,
    }
  },
}
```

Calling *setdeepattr* with a key path to a value deep in the structure will set that value. If the value or any of the objects in the key path do not exist, then a dict will be created.

```
# Overwrites the value in `d` at A.0.a, which was 1, to 3
setdeepattr(d, ('A', '0', 'a'), 3)

# Adds an entry in `d` to A.0 with the key 'c' and the value 3
setdeepattr(d, ('A', '0', 'c'), 3)

# Adds an entry in `d` with the key 'X' and the value a new dict
# Adds an entry in `d` to `X` with the key '0' and the value a new dict
# Adds an entry in `d` to `X.0` with the key 'a' and the value 0
setdeepattr(d, ('X', '0', 'a'), 0)
```

Parameters

- **d** – A dict value with nested dict attributes.
- **keys** – A list/tuple path of keys in *d* to the desired value
- **value** – The value to set at the given path *keys*.

fireant.utils.**wrap_list**(*value*, *wrapper=<class 'list'>*)

CHAPTER 2

Indices and tables

- genindex
- modindex

Python Module Index

f

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

V

W