Datadiff Documentation

Erez Shinan

Oct 20, 2022

REFERENCE

data_diff.**connect**(*db_conf: Union[str, dict]*, *thread_count: Optional[int] = 1*) → Database

Connect to a database using the given database configuration.

Configuration can be given either as a URI string, or as a dict of {option: value}.

thread_count determines the max number of worker threads per database, if relevant. None means no limit.

Parameters

- **db_conf** (*str | dict*) – The configuration for the database to connect. URI or dict.
- **thread_count** (*int, optional*) – Size of the threadpool. Ignored by cloud databases. (default: 1)

Note: For non-cloud databases, a low thread-pool size may be a performance bottleneck.

Supported drivers: - postgresql - mysql - oracle - snowflake - bigquery - redshift - presto - databricks - trino - clickhouse - vertica

class data_diff.**TableDiffer**(*bisection_factor: int = 32*, *bisection_threshold: Number = 16384*, *threaded: bool = True*, *max_threadpool_size: (NoneType+int) = 1*, *debug: bool = False*, *stats: dict[(Any*Any)] = <factory>*)

Finds the diff between two SQL tables

The algorithm uses hashing to quickly check if the tables are different, and then applies a bisection search recursively to find the differences efficiently.

Works best for comparing tables that are mostly the same, with minor discrepencies.

Parameters

- **bisection_factor** (*int*) – Into how many segments to bisect per iteration.
- **bisection_threshold** (*Number*) – When should we stop bisecting and compare locally (in row count).
- **threaded** (*bool*) – Enable/disable threaded diffing. Needed to take advantage of database threads.
- **max_threadpool_size** (*int*) – Maximum size of each threadpool. `None` means auto. Only relevant when *threaded* is `True`. There may be many pools, so number of actual threads can be a lot higher.

diff_tables(*table1:* TableSegment, *table2:* TableSegment) → Iterator[Tuple[str, tuple]]

Diff the given tables.

Parameters

- **table1** (TableSegment) – The “before” table to compare. Or: source table
- **table2** (TableSegment) – The “after” table to compare. Or: target table

Returns

An iterator that yield pair-tuples, representing the diff. Items can be either (‘-’, columns) for items in table1 but not in table2 (‘+’, columns) for items in table2 but not in table1 Where *columns* is a tuple of values for the involved columns, i.e. (id, . . . extra)

__init__(*bisection_factor: int = 32*, *bisection_threshold: Number = 16384*, *threaded: bool = True*, *max_threadpool_size: (NoneType+int) = 1*, *debug: bool = False*, *stats: dict[(Any*Any)] = <factory>*) → None

class data_diff.**TableSegment**(*database: Database = <object object>*, *table_path: tuple[str] = <object object>*, *key_column: str = <object object>*, *update_column: (NoneType+str) = None*, *extra_columns: tuple[str] = ()*, *min_key: (NoneType+(ArithUUID+ArithAlphanumeric+str+bytes+int)) = None*, *max_key: (NoneType+(ArithUUID+ArithAlphanumeric+str+bytes+int)) = None*, *min_update: (NoneType+datetime) = None*, *max_update: (NoneType+datetime) = None*, *where: (NoneType+str) = None*, *case_sensitive: bool = True*, *_schema: (NoneType+CaseAwareMapping) = None*)

Signifies a segment of rows (and selected columns) within a table

Parameters

- **database** (*Database*) – Database instance. See *connect()*
- **table_path** (*DbPath*) – Path to table in form of a tuple. e.g. *('my_dataset', 'table_name')*
- **key_column** (*str*) – Name of the key column, which uniquely identifies each row (usually id)
- **update_column** (*str, optional*) – Name of updated column, which signals that rows changed (usually updated_at or last_update)
- **extra_columns** (*Tuple[str, ...], optional*) – Extra columns to compare
- **min_key** (*DbKey*, optional) – Lowest key_column value, used to restrict the segment
- **max_key** (*DbKey*, optional) – Highest key_column value, used to restrict the segment
- **min_update** (*DbTime*, optional) – Lowest update_column value, used to restrict the segment
- **max_update** (*DbTime*, optional) – Highest update_column value, used to restrict the segment
- **where** (*str, optional*) – An additional 'where' expression to restrict the search space.
- **case_sensitive** (*bool*) – If false, the case of column names will adjust according to the schema. Default is true.

get_values() → list

Download all the relevant values of the segment from the database

choose_checkpoints(*count: int*) → List[Union[int, str, bytes, ArithUUID, ArithAlphanumeric]]

Suggests a bunch of evenly-spaced checkpoints to split by (not including start, end)

segment_by_checkpoints(*checkpoints: List[Union[int, str, bytes, ArithUUID, ArithAlphanumeric]]*) → List[*TableSegment*]

Split the current TableSegment to a bunch of smaller ones, separated by the given checkpoints

new(***kwargs*) → *TableSegment*

Using new() creates a copy of the instance using 'replace()'

count() → Tuple[int, int]

Count how many rows are in the segment, in one pass.

count_and_checksum() → Tuple[int, int]

Count and checksum the rows in the segment, in one pass.

__init__(*database: Database = <object object>*, *table_path: tuple[str] = <object object>*, *key_column: str = <object object>*, *update_column: (NoneType+str) = None*, *extra_columns: tuple[str] = ()*, *min_key: (NoneType+(ArithUUID+ArithAlphanumeric+str+bytes+int)) = None*, *max_key: (NoneType+(ArithUUID+ArithAlphanumeric+str+bytes+int)) = None*, *min_update: (NoneType+datetime) = None*, *max_update: (NoneType+datetime) = None*, *where: (NoneType+str) = None*, *case_sensitive: bool = True*, *_schema: (NoneType+CaseAwareMapping) = None*) → None

class data_diff.databases.database_types.**AbstractDatabase**

abstract quote(*s: str*)

Quote SQL name (implementation specific)

abstract to_string(*s: str*) → str

Provide SQL for casting a column to string

abstract concat(*l: List[str]*) → str

Provide SQL for concatenating a bunch of column into a string

abstract is_distinct_from(*a: str*, *b: str*) → str

Provide SQL for a comparison where NULL = NULL is true

abstract timestamp_value(*t: datetime*) → str

Provide SQL for the given timestamp value

abstract md5_to_int(*s: str*) → str

Provide SQL for computing md5 and returning an int

abstract offset_limit(*offset: Optional[int] = None*, *limit: Optional[int] = None*)

Provide SQL fragment for limit and offset inside a select

abstract select_table_schema(*path: Tuple[str, ...]*) → str

Provide SQL for selecting the table schema as (name, type, date_prec, num_prec)

abstract query_table_schema(*path: Tuple[str, ...]*) → Dict[str, tuple]

Query the table for its schema for table in 'path', and return {column: tuple} where the tuple is (table_name, col_name, type_repr, datetime_precision?, numeric_precision?, numeric_scale?)

abstract parse_table_name(*name: str*) → Tuple[str, ...]

Parse the given table name into a DbPath

abstract close()

Close connection(s) to the database instance. Querying will stop functioning.

abstract normalize_timestamp(*value: str*, *coltype: TemporalType*) → str

Creates an SQL expression, that converts 'value' to a normalized timestamp.

The returned expression must accept any SQL datetime/timestamp, and return a string.

Date format: `YYYY-MM-DD HH:mm:SS.FFFFFF`

Precision of dates should be rounded up/down according to coltype.rounds

abstract normalize_number(*value: str*, *coltype: FractionalType*) → str

Creates an SQL expression, that converts 'value' to a normalized number.

The returned expression must accept any SQL int/numeric/float, and return a string.

Floats/Decimals are expected in the format "I.P"

Where I is the integer part of the number (as many digits as necessary), and must be at least one digit (0). P is the fractional digits, the amount of which is specified with coltype.precision. Trailing zeroes may be necessary. If P is 0, the dot is omitted.

Note: We use 'precision' differently than most databases. For decimals, it's the same as `numeric_scale`, and for floats, who use binary precision, it can be calculated as `log10(2**numeric_precision)`.

abstract normalize_uuid(*value: str*, *coltype: ColType_UUID*) → str

Creates an SQL expression, that converts 'value' to a normalized uuid.

i.e. just makes sure there is no trailing whitespace.

normalize_value_by_type(*value: str*, *coltype: ColType*) → str

Creates an SQL expression, that converts 'value' to a normalized representation.

The returned expression must accept any SQL value, and return a string.

The default implementation dispatches to a method according to *coltype*:

```
TemporalType    -> normalize_timestamp()
FractionalType  -> normalize_number()
*else*          -> to_string()

(`Integer` falls in the *else* category)
```

data_diff.**DbKey**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of `Union[int, str, bytes, ArithUUID, ArithAlphanumeric]`

data_diff.**DbTime = <class 'datetime.datetime'>**

datetime(year, month, day[, hour[, minute[, second[, microsecond[,tzinfo]]]]])

The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints.

data_diff.**DbPath**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of `Tuple[str, ...]`

CHAPTER

ONE

HOW TO IMPLEMENT A NEW DATABASE DRIVER FOR DATA-DIFF

First, read through the CONTRIBUTING.md document.

Make sure data-diff is set up for development, and that all the tests pass (try to at least set it up for mysql and postgresql)

Look at the other database drivers for example and inspiration.

1.1 1. Add dependencies to `pyproject.toml`

Most new drivers will require a 3rd party library in order to connect to the database.

These dependencies should be specified in the `pyproject.toml` file, in `[tool.poetry.extras]`. Example:

```
[tool.poetry.extras]
postgresql = ["psycopg2"]
```

Then, users can install the dependencies needed for your database driver, with `pip install 'data-diff[postgresql]`.

This way, data-diff can support a wide variety of drivers, without requiring our users to install libraries that they won't use.

1.2 2. Implement database module

New database modules belong in the `data_diff/databases` directory.

1.2.1 Import on demand

Database drivers should not import any 3rd party library at the module level.

Instead, they should be imported and initialized within a function. Example:

```
from .base import import_helper

@import_helper("postgresql")
def import_postgresql():
    import psycopg2
    import psycopg2.extras
```

(continues on next page)

(continued from previous page)

```
    psycopg2.extensions.set_wait_callback(psycopg2.extras.wait_select)
    return psycopg2
```

We use the `import_helper()` decorator to provide a uniform and informative error. The string argument should be the name of the package, as written in `pyproject.toml`.

1.2.2 Choosing a base class, based on threading Model

You can choose to inherit from either `base.Database` or `base.ThreadedDatabase`.

Usually, databases with cursor-based connections, like MySQL or Postgresql, only allow one thread per connection. In order to support multithreading, we implement them by inheriting from `ThreadedDatabase`, which holds a pool of worker threads, and creates a new connection per thread.

Usually, cloud databases, such as snowflake and bigquery, open a new connection per request, and support simultaneous queries from any number of threads. In other words, they already support multithreading, so we can implement them by inheriting directly from `Database`.

1.2.3 `_query()`

All queries to the database pass through `_query()`. It takes SQL code, and returns a list of rows. Here is its signature:

```
def _query(self, sql_code: str) -> list: ...
```

For standard cursor connections, it's sufficient to implement it with a call to `base._query_conn()`, like:

::

    return _query_conn(self._conn, sql_code)

1.2.4 `select_table_schema()` / `query_table_schema()`

If your database does not have a `information_schema.columns` table, or if its structure is unusual, you may have to implement your own `select_table_schema()` function, which returns the query needed to return column information in the form of a list of tuples, where each tuple is *column_name, data_type, datetime_precision, numeric_precision, numeric_scale*.

If such a query isn't possible, you may have to implement `query_table_schema()` yourself, which extracts this information from the database, and returns it in the proper form.

If the information returned from `query_table_schema()` requires slow or error-prone post-processing, you may delay that post-processing by overriding `_process_table_schema()` and implementing it there. The method `_process_table_schema()` only gets called for the columns that will be diffed.

Documentation:

- *data_diff.databases.database_types.AbstractDatabase.select_table_schema()*
- *data_diff.databases.database_types.AbstractDatabase.query_table_schema()*

1.2.5 TYPE_CLASSES

Each database class must have a TYPE_CLASSES dictionary, which maps between the string data-type, as returned by querying the table schema, into the appropriate data-diff type class, i.e. a subclass of database_types.ColType.

1.2.6 ROUNDS_ON_PREC_LOSS

When providing a datetime or a timestamp to a database, the database may lower its precision to correspond with the target column type.

Some databases will lower precision of timestamp/datetime values by truncating them, and some by rounding them.

ROUNDS_ON_PREC_LOSS should be True if this database rounds, or False if it truncates.

1.2.7 __init__(), create_connection()

The options for the database connection will be given to the __init__() method as keywords.

If you inherit from Database, your __init__() method may create the database connection.

If you inherit from ThreadedDatabase, you should instead create the connection in the create_connection() method.

1.2.8 close()

If you inherit from Database, you will need to implement this method to close the connection yourself.

If you inherit from ThreadedDatabase, you don't have to implement this method.

Docs:

- *data_diff.databases.database_types.AbstractDatabase.close()*

1.2.9 quote(), to_string(), normalize_number(), normalize_timestamp(), md5_to_int()

These methods are used when creating queries.

They accept an SQL code fragment, and returns a new code fragment representing the appropriate computation.

For more information, read their docs:

- *data_diff.databases.database_types.AbstractDatabase.quote()*
- *data_diff.databases.database_types.AbstractDatabase.to_string()*

1.3 3. Add tests

Add your new database to the `DATABASE_TYPES` dict in `tests/test_database_types.py`

The key is the class itself, and the value is a dict of {category: [type1, type2, ...]}

Categories supported are: `int`, `datetime`, `float`, and `uuid`.

Example:

```
DATABASE_TYPES = {
    ...
    db.PostgreSQL: {
        "int": [ "int",  "bigint" ],
        "datetime": [
            "timestamp(6) without time zone",
            "timestamp(3) without time zone",
            "timestamp(0) without time zone",
            "timestamp with time zone",
        ],
        ...
    },
```

Then run the tests and make sure your database driver is being tested.

You can run the tests with `unittest`.

To save time, we recommend running them with `unittest-parallel`.

When debugging, we recommend using the *-f* flag, to stop on error. Also, use the *-k* flag to run only the individual test that you're trying to fix.

1.4 4. Create Pull-Request

Open a pull-request on github, and we'll take it from there!

CHAPTER

TWO

INTRODUCTION

Data-diff is a command-line tool and Python library to efficiently diff rows across two different databases.

- Verifies across many different databases (e.g. *PostgreSQL -> Snowflake*) !
- Outputs diff of rows in detail
- Simple CLI/API to create monitoring and alerts
- Verify 25M+ rows in <10s, and 1B+ rows in ~5min.
- Works for tables with 10s of billions of rows

For more information, See our README

CHAPTER

THREE

HOW TO INSTALL

Requires Python 3.7+ with pip.

```
pip install data-diff
```

or when you need extras like mysql and postgresql:

```
pip install "data-diff[mysql,postgresql]"
```

CHAPTER

FOUR

HOW TO USE FROM PYTHON

```python
# Optional: Set logging to display the progress of the diff
import logging
logging.basicConfig(level=logging.INFO)

from data_diff import connect_to_table, diff_tables

table1 = connect_to_table("postgresql:///", "table_name", "id")
table2 = connect_to_table("mysql:///", "table_name", "id")

for sign, columns in diff_tables(table1, table2):
    print(sign, columns)

# Example output:
+ ('4775622148347', '2022-06-05 16:57:32.000000')
- ('4775622312187', '2022-06-05 16:57:32.000000')
- ('4777375432955', '2022-06-07 16:57:36.000000')
```

CHAPTER

FIVE

RESOURCES

- Source code (git): https://github.com/datafold/data-diff
- **API Reference**
  - *Python API Reference*
- **Guides**
  - *How to implement a new database driver for data-diff*
- **Tutorials**
  - TODO

PYTHON MODULE INDEX

d

INDEX

Symbols

A

C

D

G

I

M

N

O

P

Q

S

T