Fugue

Han Wang

Aug 29, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is Fugue?

1.1.1 Fugue is a framework

It is a pure abstraction layer that adapts to different computing frameworks such as Spark and Dask. It is to unify the core concepts of distributed computing and to help you decouple your logic from specific computing frameworks.

1.1.2 Fugue is a way of thinking

It helps you rethink your problems in a platform and scale agnostic way, it helps you separate computation and orchestration. You will naturally write more of your logic in native python, and minimize the dependency on Fugue and any underlying computing frameworks in your workflow. You will find your code more testable and more portable.

1.2 Why should I consider Fugue?

1.2.1 It adapts to your code

For most cases, in order to integrate with Fugue, you only need to follow the general python type hints (which is also regarded as good practice for writing python code), and your functions can be recognized and used by Fugue. It will adapt to your input and output specs, you can write the logic in the most intuitive way (see examples)

1.2.2 It makes your logic portable

Fugue is an abstract computing layer, it adapts to different computing frameworks. So if you can express your logic purely on Fugue level, it can run on all supported computing frameworks. You are also able to prototype on local machine with small data, and then make the same workflow run on a distributed environemnt on big data. Productionizing Fugue workflows is simple and seamless.

1.2.3 It improves testability

Develop your pipelines in the Fugue way, more code especially the computation part, stays native, so unit testing them shouldn't be challenging. For the orchestration part, since you can run with `NativeExecutionEngine` with small mock data, they also become more testable. Node2Vec on Fugue is a perfect example to demonstrate the testability.

1.2.4 It's easy to onboard and offboard

To onboard, most of your code will stay native and independent from Fugue. You can simly use Fugue to *glue* your units together to be a scalable and portable workflow. On the other hand, moving away from Fugue is also easy because there is minimal code to change. Moreover, the Fugue way of thinking will be beneficial even when you migrate to other computing frameworks. You will get a better sense of layering your code and keeping it less coupled with the framework.

1.3 What does Fugue NOT do?

1.3.1 It does not hijack your project

It encourages writing native python and the framework will adapt to you. So most of your logic should stay independent. The parts related with Fugue (mostly orchestration part) should also be intuitive and readable, it shouldn't be hard to translate to other computing frameworks.

1.3.2 It does not prevent you from using the underlying frameworks

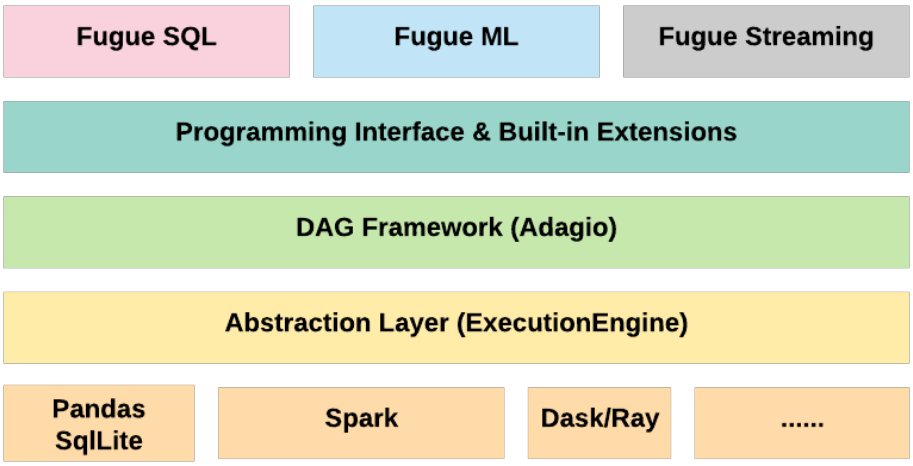
For example, if you really need to use certain features of Spark RDD, you can create an execution engine aware extension where you have full access to Spark Sessiuon and RDD, so you can write whatever native Spark code that is necessary. In general, for all computing frameworks Fugue supports, you have full access to all their features, and you can write in their native way, the only drawback is that this type of extensions is no longer portable.

1.3.3 It is not for macro-level workflows

Fugue is focusing on computataion, it's NOT another *macro-level* workflow solution such as Airflow, Prefect, Dagster and Flyte, instead, Fugue should be used by these solutions as tasks. These macro level solutions focus on orchestrating your business logic, and Fugue is for computation related logic. It's true that Fugue also uses the *DAG* concept, but it should be considered as a *micro-level* workflow solution.

1.4 Key Features

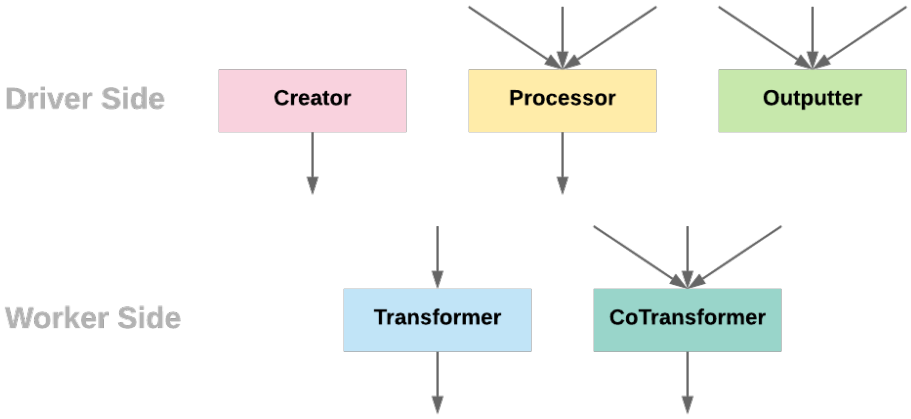
1.4.1 Architecture Overview



On top of different computing frameworks, *ExecutionEngine* is to unify the core concepts of distributed computing and adapt to different frameworks. On top of ExecutionEngine, we use Adagio to construct and execute Fugue DAGs. On top of the DAG framework, we have Fugue programming interface with built-in extensions such as *save*, load and *show*. On top of the programming interface, we have Fugue SQL and will release Fugue ML and Fugue Streaming later in 2020.

1.4.2 Extensions

Fugue extensions are the logic units you want to use Fugue to glue together. Please make sure you understand that it's NOT necessary to implement Fugue interfaces to become Fugue extensions. For details, please read the Extensions Tutorial



1.4.3 Programming Interface & SQL

Users have two ways to use Fugue: the programming interface and the SQL interface. The two are almost equivalent on features, but to adapt to different users and scenarios. The programming way is great for pythonic users and the SQL way is preferred by people who love the SQL mindset.

Fugue SQL is a SQL-like language, it's built on top of standard SQL but in the language. It's unique because it is used to describe your end-to-end workflow, and you can easily invoke all Fugue extensions inside the code. The syntax of the language is between standard SQL, json and python, it's fully compatible with standard `SELECT` statement syntax, meanwhile it tries to minimize the syntax overhead and keep the language easy to understand.

1.5 How do I get started?

1.5.1 Try Before You Install!

Before installing Fugue, you may launch a Fugue tutorial notebook environemnt on binder

But it runs slow on binder, the machine on binder isn't powerful enough for a distributed framework such as Spark. Parallel executions can become sequential, so some of the performance comparison examples will not give you the correct numbers.

Alternatively, you should get decent performance if running its docker image on your own machine:

```
docker run -p 8888:8888 fugueproject/tutorials:latest
```

1.5.2 Installation

If you only want to prototype on Fugue programming interface:

```
pip install fugue
```

If you want to use Fugue SQL as well:

```
pip install fugue[sql]
```

If you want to run on Spark:

```
pip install fugue[spark]
```

If you want to run on Dask:

```
pip install fugue[dask]
```

Many users may want to try both Spark and Fugue SQL:

```
pip install fugue[sql,spark]
```

If you want to install all extras:

```
pip install fugue[all]
```

1.5.3 What to read?

Directly reading the source code or the Fugue API docs is NOT a good idea to start. We have created tutorials for different levels of users.

For beginners, you can go through the examples without understanding everything, and you may find answers and more details inside deep dives.

For advanced users, you can go through the examples to understand what extra value Fugue can bring to you. And if interested you can go through the deep dives to get more insights.

1.5.4 git clone

If you want to start from the source code:

```
git clone https://github.com/fugue-project/fugue.git
```

CHAPTER

TWO

FUGUE TUTORIALS

To directly read the tutorials without running them:

You may launch a Fugue tutorial notebook environemnt on binder

But it runs slow on binder, the machine on binder isn't powerful enough for a distributed framework such as Spark. Parallel executions can become sequential, so some of the performance comparison examples will not give you the correct numbers.

Alternatively, you should get decent performance if running its docker image on your own machine:

```
docker run -p 8888:8888 fugueproject/tutorials:latest
```

CHAPTER

THREE

COMMUNITY

The Fugue project is not only about Fugue framework. The goal is to build the abstraction layer on different computing and machine learning frameworks so that developers and scientists can focus more on their own tasks and focus more on WHAT to do instead of HOW to do and WHERE to do. We have a very diversed open source world, many frameworks are great only on certain things but they try to create their own ecosystems excluding others. We believe that we deserve a more unified approach to distributed computing and machine learning, and we deserve the freedom of choosing different options.

3.1 Ask a question

Please join Slack chat to ask questions. We will try to reply as soon as possible.

3.2 Request bug fix or new features

You can request bug fix or new features at https://github.com/fugue-project/fugue/issues, before submitting a new request, it will be great if you can firstly contact us through slack.

3.3 Contribute

Fugue is a very new project, we truly appreciate if you can contribute code, ideas or docs. Please reach out using slack, we will be excited to chat with you.

3.3.1 Creating a development environment

There are three steps to setting-up a development environment

1. Create a virtual environment with your choice of environment manager
2. Install the requirements
3. Install the git hook scripts

Creating an environment

Below are examples for how to create and activate an environment in virtualenv and conda.

Using virtualenv

```
python3 -m venv venv
. venv/bin/activate
```

Using conda

```
conda create --name fugue-dev
conda activate fugue-dev
```

Installing requirements

The Fugue repo has a Makefile that can be used to install the requirements. It supports installation in both pip and conda. Instructions to install *make* for Windows users can be found later.

Pip install requirements

```
make setupinpip
```

Conda install requirements

```
make setupinconda
```

Manually install requirements

For Windows users who don't have the *make* command, you can use your package manager of choice. For pip:

```
pip3 install -r requirements.txt
```

For Anaconda users, first install pip in the newly created environment. If pip install is used without installing pip, conda will use the system-wide pip

```
conda install pip
pip install -r requirements.txt
```

Notes for Windows Users

For Windows users, you will need to download Microsoft C++ Build Tools found [here](https://visualstudio.microsoft.com/visual-cpp-build-tools/)

make is a GNU command that does not come with Windows. An installer can be downloaded [here](http://gnuwin32.sourceforge.net/packages/make.htm) After installing, add the bin to your PATH environment variable.

Installing git hook scripts

Fugue has pre-commit hooks to check if code is appropriate to be commited. The previous *make* command installs this. If you installed the requirements manually, install the git hook scripts with:

```
pre-commit install
```

CHAPTER

FOUR

API REFERENCE

4.1 fugue

4.1.1 fugue.collections

fugue.collections.partition

class fugue.collections.partition.**PartitionCursor**(*schema*, *spec*, *physical_partition_no*)

Bases: object

The cursor pointing at the first row of each logical partition inside a physical partition.

It's important to understand the concept of partition, please read the Partition Tutorial

Parameters

- **schema** (*triad.collections.schema.Schema*) – input dataframe schema
- **spec** (fugue.collections.partition.PartitionSpec) – partition spec
- **physical_partition_no** (*int*) – physical partition number passed in by *ExecutionEngine*

property key_schema: triad.collections.schema.Schema

Partition key schema

property key_value_array: List[Any]

Based on current row, get the partition key values as an array

property key_value_dict: Dict[str, Any]

Based on current row, get the partition key values as a dict

property partition_no: int

Logical partition number

property physical_partition_no: int

Physical partition number

property row: List[Any]

Get current row data

property row_schema: triad.collections.schema.Schema

Schema of the current row

set(*row*, *partition_no*, *slice_no*)

reset the cursor to a row (which should be the first row of a new logical partition)

Parameters

- **row** (*Any*) – list-like row data
- **partition_no** (*int*) – logical partition number
- **slice_no** (*int*) – slice number inside the logical partition (to be deprecated)

Return type None

property slice_no: int
Slice number (inside the current logical partition), for now it should always be 0

class fugue.collections.partition.**PartitionSpec**(**args*, ***kwargs*)
Bases: object

Fugue Partition Specification.

Examples

```
>>> PartitionSepc(num=4)
>>> PartitionSepc(num="ROWCOUNT/4 + 3")  # It can be an expression
>>> PartitionSepc(by=["a","b"])
>>> PartitionSpec(by=["a"], presort="b DESC, c ASC")
>>> PartitionSpec(algo="even", num=4)
>>> p = PartitionSpec(num=4, by=["a"])
>>> p_override = PartitionSpec(p, by=["a","b"], algo="even")
>>> PartitionSpec(by="a")  # == PartitionSpec(by=["a"])
>>> PartitionSpec("per_row")  # == PartitionSpec(num="ROWCOUNT", algo="even")
```

It's important to understand this concept, please read the Partition Tutorial

Partition consists for these specs:

- **algo**: can be one of `hash` (default), `rand` and `even`
- **num** or **num_partitions**: number of physical partitions, it can be an expression or integer numbers, e.g `(ROWCOUNT+4) / 3`
- **by** or **partition_by**: keys to partition on
- **presort**: keys to sort other than partition keys. E.g. `a` and `a asc` means presort by column a ascendingly, `a,b desc` means presort by a ascendingly and then by b descendingly.
- row_limit and size_limit are to be deprecated

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

property algo: str
Get algo of the spec, one of `hash` (default), `rand` and `even`

property empty: bool
Whether this spec didn't specify anything

get_cursor(*schema*, *physical_partition_no*)
Get *PartitionCursor* based on dataframe schema and physical partition number. You normally don't call this method directly

Parameters

- **schema** (*triad.collections.schema.Schema*) – the dataframe schema this partition spec to operate on
- **physical_partition_no** (*int*) – physical partition no passed in by *ExecutionEngine*

Returns PartitionCursor object

Return type *fugue.collections.partition.PartitionCursor*

get_key_schema(*schema*)

Get partition keys schema

Parameters **schema** (*triad.collections.schema.Schema*) – the dataframe schema this partition spec to operate on

Returns the sub-schema only containing partition keys

Return type triad.collections.schema.Schema

get_num_partitions(***expr_map_funcs*)

Convert `num_partitions` expression to int number

Parameters **expr_map_funcs** (*Any*) – lambda functions (no parameter) for keywords

Returns integer value of the partitions

Return type int

Examples

```
>>> p = PartitionSpec(num="ROWCOUNT/2")
>>> p.get_num_partitions(ROWCOUNT=lambda: df.count())
```

---

get_partitioner(*schema*)

Get `SchemaedDataPartitioner` by input dataframe schema

Parameters **schema** (*triad.collections.schema.Schema*) – the dataframe schema this partition spec to operate on

Returns SchemaedDataPartitioner object

Return type triad.utils.pyarrow.SchemaedDataPartitioner

get_sorts(*schema*)

Get keys for sorting in a partition, it's the combination of partition keys plus the presort keys

Parameters **schema** (*triad.collections.schema.Schema*) – the dataframe schema this partition spec to operate on

Returns an ordered dictionary of key, order pairs

Return type triad.collections.dict.IndexedOrderedDict[str, bool]

Examples

```
>>> p = PartitionSpec(by=["a"],presort="b , c dESc")
>>> schema = Schema("a:int,b:int,c:int,d:int"))
>>> assert p.get_sorts(schema) == {"a":True, "b":True, "c": False}
```

---

property jsondict: triad.collections.dict.ParamDict
Get json serializeable dict of the spec

property num_partitions: str
Number of partitions, it can be a string expression or int

property partition_by: List[str]
Get partition keys of the spec

property presort: triad.collections.dict.IndexedOrderedDict[str, bool]
Get presort pairs of the spec

Examples

```
>>> p = PartitionSpec(by=["a"],presort="b,c desc")
>>> assert p.presort == {"b":True, "c":False}
```

property presort_expr: str
Get normalized presort expression

Examples

```
>>> p = PartitionSpec(by=["a"],presort="b , c dESc")
>>> assert p.presort_expr == "b ASC,c DESC"
```

fugue.collections.partition.**parse_presort_exp**(*presort*)
Returns ordered column sorting direction where ascending order would return as true, and descending as false.

Parameters **presort** (*Any*) – string that contains column and sorting direction or list of tuple that contains column and boolean sorting direction

Returns column and boolean sorting direction of column, order matters.

Return type IndexedOrderedDict[str, bool]

Examples

```
>>> parse_presort_exp("b desc, c asc")
>>> parse_presort_exp([("b", True), ("c", False)])
both return IndexedOrderedDict([("b", True), ("c", False)])
```

fugue.collections.yielded

class fugue.collections.yielded.**Yielded**(*yid*)
Bases: object

Yields from *FugueWorkflow*. Users shouldn't create this object directly.

Parameters **yid** (*str*) – unique id for determinism

property is_set: bool
Whether the value is set. It can be false if the parent workflow has not been executed.

class fugue.collections.yielded.**YieldedFile**(*yid*)
Bases: *fugue.collections.yielded.Yielded*

Yielded file from *FugueWorkflow*. Users shouldn't create this object directly.

Parameters **yid** (*str*) – unique id for determinism

property **is_set: bool**
Whether the value is set. It can be false if the parent workflow has not been executed.

property **path: str**
File path of the yield

set_value(*path*)
Set the yielded path after compute

Parameters **path** (*str*) – file path

Return type None

4.1.2 fugue.column

fugue.column.expressions

class fugue.column.expressions.**ColumnExpr**
Bases: object

Fugue column expression class. It is inspired from pyspark.sql.Column and it is working in progress.

New Since

0.6.0

Caution: This is a base class of different column classes, and users are not supposed to construct this class directly. Use *col()* and *lit()* instead.

alias(*as_name*)
Assign or remove alias of a column. To remove, set as_name to empty

Returns a new column with the alias value

Parameters **as_name** (*str*) –

Return type *fugue.column.expressions.ColumnExpr*

Examples

```
assert "b" == col("a").alias("b").as_name
assert "x" == (col("a") * 2).alias("x").as_name
assert "" == col("a").alias("b").alias("").as_name
```

property **as_name: str**
The name assigned by *alias()*

Returns the alias

Examples

```
assert "" == col("a").as_name
assert "b" == col("a").alias("b").as_name
assert "x" == (col("a") * 2).alias("x").as_name
```

property as_type: Optional[pyarrow.lib.DataType]

The type assigned by *cast()*

Returns the pyarrow datatype if *cast()* was called otherwise None

Examples

```
import pyarrow as pa

assert col("a").as_type is None
assert pa.int64() == col("a").cast(int).as_type
assert pa.string() == (col("a") * 2).cast(str).as_type
```

property body_str: str

The string expression of this column without cast type and alias. This is only used for debug purpose. It is not SQL expression.

Returns the string expression

cast(*data_type*)

Cast the column to a new data type

Parameters **data_type** (*Any*) – It can be string expressions, python primitive types, python *datetime.datetime* and pyarrow types. For details read Fugue Data Types

Returns a new column instance with the assigned data type

Return type *fugue.column.expressions.ColumnExpr*

Caution: Currently, casting to struct or list type has undefined behavior.

Examples

```
import pyarrow as pa

assert pa.int64() == col("a").cast(int).as_type
assert pa.string() == col("a").cast(str).as_type
assert pa.float64() == col("a").cast(float).as_type
assert pa._bool() == col("a").cast(bool).as_type

# string follows the type expression of Triad Schema
assert pa.int32() == col("a").cast("int").as_type
assert pa.int32() == col("a").cast("int32").as_type

assert pa.int32() == col("a").cast(pa.int32()).as_type
```

infer_alias()

Infer alias of a column. If the column's *output_name()* is not empty then it returns itself without change. Otherwise it tries to infer alias from the underlying columns.

Returns a column instance with inferred alias

Return type *fugue.column.expressions.ColumnExpr*

Caution: Users should not use it directly.

Examples

```
import fugue.column.functions as f

assert "a" == col("a").infer_alias().output_name
assert "" == (col("a") * 2).infer_alias().output_name
assert "a" == col("a").is_null().infer_alias().output_name
assert "a" == f.max(col("a").is_null()).infer_alias().output_name
```

infer_type(*schema*)

Infer data type of this column given the input schema

Parameters **schema** (*triad.collections.schema.Schema*) – the schema instance to infer from

Returns a pyarrow datatype or None if failed to infer

Return type Optional[pyarrow.lib.DataType]

Caution: Users should not use it directly.

Examples

```
import pyarrow as pa
from triad import Schema
import fugue.column.functions as f

schema = Schema("a:int,b:str")

assert pa.int32() == col("a").infer_schema(schema)
assert pa.int32() == (-col("a")).infer_schema(schema)
# due to overflow risk, can't infer certain operations
assert (col("a")+1).infer_schema(schema) is None
assert (col("a")+col("a")).infer_schema(schema) is None
assert pa.int32() == f.max(col("a")).infer_schema(schema)
assert pa.int32() == f.min(col("a")).infer_schema(schema)
assert f.sum(col("a")).infer_schema(schema) is None
```

is_null()
Same as SQL <col> IS NULL.

Returns a new column with the boolean values

Return type *fugue.column.expressions.ColumnExpr*

property name: str
The original name of this column, default empty

Returns the name

Examples

```
assert "a" == col("a").name
assert "b" == col("a").alias("b").name
assert "" == lit(1).name
assert "" == (col("a") * 2).name
```

---

not_null()
Same as SQL <col> IS NOT NULL.

Returns a new column with the boolean values

Return type *fugue.column.expressions.ColumnExpr*

property output_name: str
The name assigned by *alias()*, but if empty then return the original column name

Returns the alias or the original column name

Examples

```
assert "a" == col("a").output_name
assert "b" == col("a").alias("b").output_name
assert "x" == (col("a") * 2).alias("x").output_name
```

---

fugue.column.expressions.**col**(*obj*, *alias=''*)
Convert the obj to a *ColumnExpr* object

Parameters

- **obj** (*Union[str, fugue.column.expressions.ColumnExpr]*) – a string representing a column name or a *ColumnExpr* object
- **alias** (*str*) – the alias of this column, defaults to "" (no alias)

Returns a literal column expression

Return type *fugue.column.expressions.ColumnExpr*

New Since

0.6.0

Examples

```
import fugue.column import col
import fugue.column.functions as f

col("a")
col("a").alias("x")
col("a", "x")

# unary operations
-col("a")  # negative value of a
~col("a")  # NOT a
col("a").is_null()  # a IS NULL
col("a").not_null()  # a IS NOT NULL

# binary operations
col("a") + 1  # col("a") + lit(1)
1 - col("a")  # lit(1) - col("a")
col("a") * col("b")
col("a") / col("b")

# binary boolean expressions
col("a") == 1  # col("a") == lit(1)
2 != col("a")  # col("a") != lit(2)
col("a") < 5
col("a") > 5
col("a") <= 5
col("a") >= 5
(col("a") < col("b")) & (col("b") > 1) | col("c").is_null()

# with functions
f.max(col("a"))
f.max(col("a")+col("b"))
f.max(col("a")) + f.min(col("b"))
f.count_distinct(col("a")).alias("dcount")
```

---

fugue.column.expressions.**function**(*name*, **args*, *arg_distinct=False*, ***kwargs*)

Construct a function expression

Parameters

- **name** (*str*) – the name of the function
- **arg_distinct** (*bool*) – whether to add `DISTINCT` before all arguments, defaults to False
- **args** (*Any*) –

Returns the function expression

Return type *fugue.column.expressions.ColumnExpr*

> **Caution:** Users should not use this directly

fugue.column.expressions.**lit**(*obj*, *alias=''*)

Convert the `obj` to a literal column. Currently `obj` must be `int`, `bool`, `float` or `str`, otherwise an exception will be raised

Parameters

- **obj** (*Any*) – an arbitrary value
- **alias** (*str*) – the alias of this literal column, defaults to "" (no alias)

Returns a literal column expression

Return type *fugue.column.expressions.ColumnExpr*

New Since

0.6.0

Examples

```
import fugue.column import lit

lit("abc")
lit(100).alias("x")
lit(100, "x")
```

---

fugue.column.expressions.**null**()

Equivalent to `lit(None)`, the `NULL` value

Returns `lit(None)`

Return type *fugue.column.expressions.ColumnExpr*

New Since

0.6.0

---

fugue.column.functions

fugue.column.functions.**avg**(*col*)

SQL `AVG` function (aggregation)

Parameters **col** (`fugue.column.expressions.ColumnExpr`) – the column to find average

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function cannot infer type from `col` type
- this function can infer alias from `col`'s inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

f.avg(col("a"))  # AVG(a) AS a

# you can specify explicitly
# CAST(AVG(a) AS double) AS a
f.avg(col("a")).cast(float)
```

fugue.column.functions.**coalesce**(**args*)

SQL COALESCE function

Parameters **args** (*Any*) – If a value is not *ColumnExpr* then it's converted to a literal column by *col()*

Return type *fugue.column.expressions.ColumnExpr*

Note: this function can infer neither type nor alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

f.coalesce(col("a"), col("b")+col("c"), 1)
```

fugue.column.functions.**count**(*col*)

SQL COUNT function (aggregation)

Parameters **col** (fugue.column.expressions.ColumnExpr) – the column to find count

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function cannot infer type from col type
- this function can infer alias from col's inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

f.count(col("*"))  # COUNT(*)
f.count(col("a"))  # COUNT(a) AS a

# you can specify explicitly
# CAST(COUNT(a) AS double) AS a
f.count(col("a")).cast(float)
```

---

fugue.column.functions.**count_distinct**(*col*)

SQL COUNT DISTINCT function (aggregation)

Parameters **col** (fugue.column.expressions.ColumnExpr) – the column to find distinct element count

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function cannot infer type from col type
- this function can infer alias from col's inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

f.count_distinct(col("*"))  # COUNT(DISTINCT *)
f.count_distinct(col("a"))  # COUNT(DISTINCT a) AS a

# you can specify explicitly
# CAST(COUNT(DISTINCT a) AS double) AS a
f.count_distinct(col("a")).cast(float)
```

---

fugue.column.functions.**first**(*col*)

SQL FIRST function (aggregation)

Parameters **col** (fugue.column.expressions.ColumnExpr) – the column to find first

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function can infer type from col type
- this function can infer alias from col's inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

# assume col a has type double
f.first(col("a"))  # CAST(FIRST(a) AS double) AS a
f.first(-col("a"))  # CAST(FIRST(-a) AS double) AS a

# neither type nor alias can be inferred in the following cases
f.first(col("a")+1)
f.first(col("a")+col("b"))

# you can specify explicitly
# CAST(FIRST(a+b) AS int) AS x
f.first(col("a")+col("b")).cast(int).alias("x")
```

fugue.column.functions.**is_agg**(*column*)

Check if a column contains aggregation operation

Parameters

- **col** – the column to check
- **column** (*Any*) –

Returns whether the column is *ColumnExpr* and contains aggregation operations

Return type bool

New Since

0.6.0

Examples

```
import fugue.column.functions as f

assert not f.is_agg(1)
assert not f.is_agg(col("a"))
assert not f.is_agg(col("a")+lit(1))

assert f.is_agg(f.max(col("a")))
assert f.is_agg(-f.max(col("a")))
assert f.is_agg(f.max(col("a")+1))
assert f.is_agg(f.max(col("a"))+f.min(col("a")))
```

fugue.column.functions.**last**(*col*)

SQL LAST function (aggregation)

Parameters **col** (fugue.column.expressions.ColumnExpr) – the column to find last

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function can infer type from col type
- this function can infer alias from col's inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

# assume col a has type double
f.last(col("a"))  # CAST(LAST(a) AS double) AS a
f.last(-col("a"))  # CAST(LAST(-a) AS double) AS a

# neither type nor alias can be inferred in the following cases
f.last(col("a")+1)
f.last(col("a")+col("b"))

# you can specify explicitly
# CAST(LAST(a+b) AS int) AS x
f.last(col("a")+col("b")).cast(int).alias("x")
```

fugue.column.functions.**max**(*col*)

SQL MAX function (aggregation)

Parameters **col** (fugue.column.expressions.ColumnExpr) – the column to find max

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function can infer type from col type
- this function can infer alias from col's inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

# assume col a has type double
f.max(col("a"))  # CAST(MAX(a) AS double) AS a
f.max(-col("a"))  # CAST(MAX(-a) AS double) AS a

# neither type nor alias can be inferred in the following cases
f.max(col("a")+1)
f.max(col("a")+col("b"))

# you can specify explicitly
# CAST(MAX(a+b) AS int) AS x
f.max(col("a")+col("b")).cast(int).alias("x")
```

---

fugue.column.functions.**min**(*col*)

SQL MIN function (aggregation)

Parameters **col** (fugue.column.expressions.ColumnExpr) – the column to find min

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function can infer type from col type
- this function can infer alias from col's inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

# assume col a has type double
f.min(col("a"))  # CAST(MIN(a) AS double) AS a
f.min(-col("a"))  # CAST(MIN(-a) AS double) AS a

# neither type nor alias can be inferred in the following cases
f.min(col("a")+1)
f.min(col("a")+col("b"))

# you can specify explicitly
# CAST(MIN(a+b) AS int) AS x
f.min(col("a")+col("b")).cast(int).alias("x")
```

---

fugue.column.functions.**sum**(*col*)

SQL SUM function (aggregation)

Parameters **col** (fugue.column.expressions.ColumnExpr) – the column to find sum

Return type *fugue.column.expressions.ColumnExpr*

Note:

- this function cannot infer type from `col` type
- this function can infer alias from `col`'s inferred alias

New Since

0.6.0

Examples

```
import fugue.column.functions as f

f.sum(col("a"))  # SUM(a) AS a

# you can specify explicitly
# CAST(SUM(a) AS double) AS a
f.sum(col("a")).cast(float)
```

---

fugue.column.sql

class fugue.column.sql.**SQLExpressionGenerator**(*enable_cast=True*)

Bases: `object`

SQL generator for *SelectColumns*

Parameters **enable_cast** (*bool*) – whether convert `cast` into the statement, defaults to True

---

New Since

0.6.0

---

add_func_handler(*name*, *handler*)

Add special function handler.

Parameters

- **name** (*str*) – name of the function
- **handler** (*Callable[[fugue.column.expressions._FuncExpr], Iterable[str]]*) – the function to convert the function expression to SQL clause

Returns the instance itself

Return type *fugue.column.sql.SQLExpressionGenerator*

Caution: Users should not use this directly

correct_select_schema(*input_schema*, *select*, *output_schema*)
Do partial schema inference from `input_schema` and `select` columns, then compare with the SQL output dataframe schema, and return the different part as a new schema, or None if there is no difference

Parameters

- **input_schema** (*triad.collections.schema.Schema*) – input dataframe schema for the select statement
- **select** (fugue.column.sql.SelectColumns) – the collection of select columns
- **output_schema** (*triad.collections.schema.Schema*) – schema of the output dataframe after executing the SQL

Returns the difference as a new schema or None if no difference

Return type Optional[triad.collections.schema.Schema]

Tip: This is particularly useful when the SQL engine messed up the schema of the output. For example, `SELECT *` should return a dataframe with the same schema of the input. However, for example a column `a:int` could become `a:long` in the output dataframe because of information loss. This function is designed to make corrections on column types when they can be inferred. This may not be perfect but it can solve major discrepancies.

---

generate(*expr*)
Convert *ColumnExpr* to SQL clause

Parameters **expr** (fugue.column.expressions.ColumnExpr) – the column expression to convert

Returns the SQL clause for this expression

Return type str

select(*columns*, *table*, *where=None*, *having=None*)
Construct the full `SELECT` statement on a single table

Parameters

- **columns** (fugue.column.sql.SelectColumns) – columns to select, it may contain aggregations, if so, the group keys are inferred. See *group_keys()*
- **table** (*str*) – table name to select from
- **where** (*Optional[fugue.column.expressions.ColumnExpr]*) – `WHERE` condition, defaults to None
- **having** (*Optional[fugue.column.expressions.ColumnExpr]*) – `HAVING` condition, defaults to None. It is used only when there is aggregation

Returns the full `SELECT` statement

Return type str

type_to_expr(*data_type*)

Parameters **data_type** (*pyarrow.lib.DataType*) –

where(*condition*, *table*)
Generate a `SELECT *` statement with the given where clause

Parameters

- **condition** (fugue.column.expressions.ColumnExpr) – column expression for WHERE
- **table** (*str*) – table name for FROM

Returns the SQL statement

Raises **ValueError** – if `condition` contains aggregation

Return type str

Examples

```
gen = SQLExpressionGenerator(enable_cast=False)

# SELECT * FROM tb WHERE a>1 AND b IS NULL
gen.where((col("a")>1) & col("b").is_null(), "tb")
```

---

class fugue.column.sql.**SelectColumns**(**cols*, *arg_distinct=False*)

Bases: `object`

SQL `SELECT` columns collection.

Parameters

- **cols** (fugue.column.expressions.ColumnExpr) – collection of *ColumnExpr*
- **arg_distinct** (*bool*) – whether this is `SELECT DISTINCT`, defaults to False

---

New Since

0.6.0

---

property agg_funcs: List[*fugue.column.expressions.ColumnExpr*]

All columns with aggregation operations

property all_cols: List[*fugue.column.expressions.ColumnExpr*]

All columns (with inferred aliases)

assert_all_with_names()

Assert every column have explicit alias or the alias can be inferred (non empty value). It will also validate there is no duplicated aliases

Raises **ValueError** – if there are columns without alias, or there are duplicated names.

Returns the instance itself

Return type *fugue.column.sql.SelectColumns*

assert_no_agg()

Assert there is no aggregation operation on any column.

Raises **AssertionError** – if there is any aggregation in the collection.

Returns the instance itself

Return type *fugue.column.sql.SelectColumns*

See also:

Go to *is_agg()* to see how the aggregations are detected.

assert_no_wildcard()
Assert there is no * on first level columns

Raises **AssertionError** – if `col("*")` exists

Returns the instance itself

Return type *fugue.column.sql.SelectColumns*

property group_keys: List[*fugue.column.expressions.ColumnExpr*]
Group keys inferred from the columns.

Note:

- if there is no aggregation, the result will be empty
- it is *simple_cols()* plus *non_agg_funcs()*

---

property has_agg: bool
Whether this select is an aggregation

property has_literals: bool
Whether this select contains literal columns

property is_distinct: bool
Whether this is a `SELECT DISTINCT`

property literals: List[*fugue.column.expressions.ColumnExpr*]
All literal columns

property non_agg_funcs: List[*fugue.column.expressions.ColumnExpr*]
All columns with non-aggregation operations

replace_wildcard(*schema*)
Replace wildcard * with explicit column names

Parameters **schema** (*triad.collections.schema.Schema*) – the schema used to parse the wildcard

Returns a new instance containing only explicit columns

Return type *fugue.column.sql.SelectColumns*

Note: It only replaces the top level *. For example `count_distinct(col("*"))` will not be transformed because this * is not first level.

---

property simple: bool
Whether this select contains only simple column representations

property simple_cols: List[*fugue.column.expressions.ColumnExpr*]
All columns directly representing column names

4.1.3 fugue.dataframe

fugue.dataframe.array_dataframe

class fugue.dataframe.array_dataframe.**ArrayDataFrame**(*df=None*, *schema=None*, *metadata=None*)

Bases: *fugue.dataframe.dataframe.LocalBoundedDataFrame*

DataFrame that wraps native python 2-dimensional arrays. Please read the DataFrame Tutorial to understand the concept

Parameters

- **df** (*Any*) – 2-dimensional array, iterable of arrays, or *DataFrame*
- **schema** (*Any*) – Schema like object
- **metadata** (*Any*) – dict-like object with string keys, default None

---

Examples

```
>>> a = ArrayDataFrame([[0,'a'],[1,'b']],"a:int,b:str")
>>> b = ArrayDataFrame(a)
```

---

alter_columns(*columns*)

Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)

Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If type_safe is False, then the returned values are 'raw' values.

---

as_array_iterable(*columns=None*, *type_safe=False*)

Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

count()
Get number of rows of this dataframe

Return type int

property empty: bool
Whether this dataframe is empty

property native: List[Any]
2-dimensional native python array

peek_array()
Peek the first row of the dataframe as array

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Any

rename(*columns*)
Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

fugue.dataframe.arrow_dataframe

class fugue.dataframe.arrow_dataframe.**ArrowDataFrame**(*df=None*, *schema=None*, *metadata=None*, *pandas_df_wrapper=False*)

Bases: *fugue.dataframe.dataframe.LocalBoundedDataFrame*

DataFrame that wraps `pyarrow.Table`. Please also read the DataFrame Tutorial to understand this Fugue concept

Parameters

- **df** (*Any*) – 2-dimensional array, iterable of arrays, `pyarrow.Table` or pandas DataFrame
- **schema** (*Any*) – Schema like object
- **metadata** (*Any*) – dict-like object with string keys, default `None`
- **pandas_df_wrapper** (*bool*) –

Examples

```
>>> ArrowDataFrame([[0,'a'],[1,'b']],"a:int,b:str")
>>> ArrowDataFrame(schema = "a:int,b:int")  # empty dataframe
>>> ArrowDataFrame(pd.DataFrame([[0]],columns=["a"]))
>>> ArrowDataFrame(ArrayDataFrame([[0]],"a:int).as_arrow())
```

alter_columns(*columns*)
Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)
Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

---

as_array_iterable(*columns=None*, *type_safe=False*)
Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

---

as_arrow(*type_safe=False*)
Convert to pyArrow DataFrame

Parameters **type_safe** (*bool*) –

Return type pyarrow.lib.Table

as_pandas()
Convert to pandas DataFrame

Return type pandas.core.frame.DataFrame

count()
Get number of rows of this dataframe

Return type int

property empty: bool
Whether this dataframe is empty

property native: pyarrow.lib.Table
pyarrow.Table

peek_array()
Peek the first row of the dataframe as array

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Any

peek_dict()
Peek the first row of the dataframe as dict

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Dict[str, Any]

rename(*columns*)
Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

fugue.dataframe.dataframe

class fugue.dataframe.dataframe.**DataFrame**(*schema=None*, *metadata=None*)
Bases: abc.ABC

Base class of Fugue DataFrame. Please read the DataFrame Tutorial to understand the concept

Parameters

- **schema** (*Any*) – Schema like object
- **metadata** (*Any*) – dict-like object with string keys, default None

---

Note: This is an abstract class, and normally you don't construct it by yourself unless you are implementing a new *ExecutionEngine*

---

abstract alter_columns(*columns*)
Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

abstract as_array(*columns=None*, *type_safe=False*)
Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If type_safe is False, then the returned values are 'raw' values.

---

abstract as_array_iterable(*columns=None*, *type_safe=False*)
Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

---

as_arrow(*type_safe=False*)
Convert to pyArrow DataFrame

Parameters **type_safe** (*bool*) –

Return type pyarrow.lib.Table

as_dict_iterable(*columns=None*)
Convert to iterable of native python dicts

Parameters **columns** (*Optional[List[str]]*) – columns to extract, defaults to None

Returns iterable of native python dicts

Return type Iterable[Dict[str, Any]]

Note: The default implementation enforces `type_safe` True

---

abstract as_local()
Convert this dataframe to a *LocalDataFrame*

Return type *fugue.dataframe.dataframe.LocalDataFrame*

as_pandas()
Convert to pandas DataFrame

Return type pandas.core.frame.DataFrame

assert_not_empty()
Assert this dataframe is not empty

Raises ***FugueDataFrameEmptyError*** – if it is empty

Return type None

abstract count()
Get number of rows of this dataframe

Return type int

drop(*columns*)
Drop certain columns and return a new dataframe

Parameters **columns** (*List[str]*) – columns to drop

Raises ***FugueDataFrameOperationError*** – if `columns` are not strictly contained by this dataframe, or it is the entire dataframe columns

Returns a new dataframe removing the columns

Return type *fugue.dataframe.dataframe.DataFrame*

abstract property empty: bool
Whether this dataframe is empty

get_info_str()
Get dataframe information (schema, type, metadata) as json string

Returns json string

Return type str

head(*n*, *columns=None*)
Get first n rows of the dataframe as 2-dimensional array

Parameters

- **n** (*int*) – number of rows
- **columns** (*Optional[List[str]]*) – selected columns, defaults to None (all columns)

Returns 2-dimensional array

Return type List[Any]

abstract property is_bounded: bool
Whether this dataframe is bounded

property is_local: bool
Whether this dataframe is a *LocalDataFrame*

property metadata: triad.collections.dict.ParamDict
Metadata of the dataframe

abstract property num_partitions: int
Number of physical partitions of this dataframe. Please read the Partition Tutorial

abstract peek_array()
Peek the first row of the dataframe as array

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Any

peek_dict()
Peek the first row of the dataframe as dict

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Dict[str, Any]

abstract rename(*columns*)
Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

property schema: triad.collections.schema.Schema
Schema of the dataframe

show(*rows=10*, *show_count=False*, *title=None*, *best_width=100*)
Print the dataframe to console

Parameters

- **rows** (*int*) – number of rows to print, defaults to 10
- **show_count** (*bool*) – whether to show dataframe count, defaults to False
- **title** (*Optional[str]*) – title of the dataframe, defaults to None
- **best_width** (*int*) – max width of the output table, defaults to 100

Return type None

Note: When show_count is True, it can trigger expensive calculation for a distributed dataframe. So if you call this function directly, you may need to *fugue.execution.execution_engine.ExecutionEngine.persist()* the dataframe.

---

class fugue.dataframe.dataframe.**LocalBoundedDataFrame**(*schema=None*, *metadata=None*)
Bases: *fugue.dataframe.dataframe.LocalDataFrame*

Base class of all local bounded dataframes. Please read this to understand the concept

Parameters

- **schema** (*Any*) – Schema like object
- **metadata** (*Any*) – dict-like object with string keys, default None

---

Note: This is an abstract class, and normally you don't construct it by yourself unless you are implementing a new *ExecutionEngine*

---

property is_bounded: bool
Always True because it's a bounded dataframe

class fugue.dataframe.dataframe.**LocalDataFrame**(*schema=None*, *metadata=None*)
Bases: *fugue.dataframe.dataframe.DataFrame*

Base class of all local dataframes. Please read this to understand the concept

Parameters

- **schema** (*Any*) – a schema-like object
- **metadata** (*Any*) – dict-like object with string keys, default None

---

Note: This is an abstract class, and normally you don't construct it by yourself unless you are implementing a new *ExecutionEngine*

---

as_local()
Always return self, because it's a LocalDataFrame

Return type *fugue.dataframe.dataframe.LocalDataFrame*

property is_local: bool
Always True because it's a LocalDataFrame

property num_partitions: int
Always 1 because it's a LocalDataFrame

class fugue.dataframe.dataframe.**LocalUnboundedDataFrame**(*schema=None*, *metadata=None*)
Bases: *fugue.dataframe.dataframe.LocalDataFrame*

Base class of all local unbounded dataframes. Read this <https://fugue-tutorials.readthedocs.io/en/latest/tutorials/advanced/schema_dataframes.html#DataFrame>`_ to understand the concept

Parameters

- **schema** (*Any*) – Schema like object
- **metadata** (*Any*) – dict-like object with string keys, default None

---

Note: This is an abstract class, and normally you don't construct it by yourself unless you are implementing a new *ExecutionEngine*

---

count()

Raises **InvalidOperationError** – You can't count an unbounded dataframe

Return type int

property is_bounded
Always False because it's an unbounded dataframe

class fugue.dataframe.dataframe.**YieldedDataFrame**(*yid*)
Bases: *fugue.collections.yielded.Yielded*

Yielded dataframe from *FugueWorkflow*. Users shouldn't create this object directly.

Parameters **yid** (*str*) – unique id for determinism

property is_set: bool
Whether the value is set. It can be false if the parent workflow has not been executed.

property result: *fugue.dataframe.dataframe.DataFrame*
The yielded dataframe, it will be set after the parent workflow is computed

set_value(*df*)
Set the yielded dataframe after compute. Users should not call it.

Parameters

- **path** – file path
- **df** (fugue.dataframe.dataframe.DataFrame) –

Return type None

fugue.dataframe.dataframe_iterable_dataframe

class fugue.dataframe.dataframe_iterable_dataframe.**LocalDataFrameIterableDataFrame**(*df=None*, *schema=None*, *metadata=None*)
Bases: *fugue.dataframe.dataframe.LocalUnboundedDataFrame*

DataFrame that wraps an iterable of local dataframes

Parameters

- **df** (*Any*) – an iterable of *DataFrame*. If any is not local, they will be converted to *LocalDataFrame* by *as_local()*
- **schema** (*Any*) – Schema like object, if it is provided, it must match the schema of the dataframes
- **metadata** (*Any*) – dict-like object with string keys, default `None`

Examples

```
def get_dfs(seq):
    yield IterableDataFrame([], "a:int,b:int")
    yield IterableDataFrame([[1, 10]], "a:int,b:int")
    yield ArrayDataFrame([], "a:int,b:str")

df = LocalDataFrameIterableDataFrame(get_dfs())
for subdf in df.native:
    subdf.show()
```

Note: It's ok to peek the dataframe, it will not affect the iteration, but it's invalid to count.

`schema` can be used when the iterable contains no dataframe. But if there is any dataframe, `schema` must match the schema of the dataframes.

For the iterable of dataframes, if there is any empty dataframe, they will be skipped and their schema will not matter. However, if all dataframes in the interable are empty, then the last empty dataframe will be used to set the schema.

alter_columns(*columns*)

Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)

Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

as_array_iterable(*columns=None*, *type_safe=False*)

Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

---

as_arrow(*type_safe=False*)
Convert to pyArrow DataFrame

Parameters **type_safe** (*bool*) –

Return type pyarrow.lib.Table

as_pandas()
Convert to pandas DataFrame

Return type pandas.core.frame.DataFrame

property empty: bool
Whether this dataframe is empty

property native:
triad.utils.iter.EmptyAwareIterable[*fugue.dataframe.dataframe.LocalDataFrame*]
Iterable of dataframes

peek_array()
Peek the first row of the dataframe as array

Raises ***FugueDataFrameEmptyError*** – if it is empty

Return type Any

rename(*columns*)
Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

fugue.dataframe.dataframes

class fugue.dataframe.dataframes.**DataFrames**(**args*, ***kwargs*)
Bases: triad.collections.dict.IndexedOrderedDict[str, *fugue.dataframe.dataframe.DataFrame*]

Ordered dictionary of DataFrames. There are two modes: with keys and without keys. If without key _<n> will be used as the key for each dataframe, and it will be treated as an array in Fugue framework.

It's a subclass of dict, so it supports all dict operations. It's also ordered, so you can trust the order of keys and values.

The initialization is flexible

```
>>> df1 = ArrayDataFrame([[0]],"a:int")
>>> df2 = ArrayDataFrame([[1]],"a:int")
>>> dfs = DataFrames(df1,df2)  # init as [df1, df2]
>>> assert not dfs.has_key
>>> assert df1 is dfs[0] and df2 is dfs[1]
>>> dfs_array = list(dfs.values())
>>> dfs = DataFrames(a=df1,b=df2)  # init as {a:df1, b:df2}
>>> assert dfs.has_key
>>> assert df1 is dfs[0] and df2 is dfs[1]  # order is guaranteed
>>> df3 = ArrayDataFrame([[1]],"b:int")
>>> dfs2 = DataFrames(dfs, c=df3)  # {a:df1, b:df2, c:df3}
>>> dfs2 = DataFrames(dfs, df3)  # invalid, because dfs has key, df3 doesn't
>>> dfs2 = DataFrames(dict(a=df1,b=df2))  # init as {a:df1, b:df2}
>>> dfs2 = DataFrames([df1,df2],df3)  # init as [df1, df2, df3]
```

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

convert(*func*)

Create another DataFrames with the same structure, but all converted by `func`

Returns the new DataFrames

Parameters **func** (*Callable[[fugue.dataframe.dataframe.DataFrame], fugue.dataframe.dataframe.DataFrame]*) –

Return type *fugue.dataframe.dataframes.DataFrames*

Examples

```
>>> dfs2 = dfs.convert(lambda df: df.as_local()) # convert all to local
```

---

property has_key

If this collection has key (dict-like) or not (list-like)

fugue.dataframe.iterable_dataframe

class fugue.dataframe.iterable_dataframe.**IterableDataFrame**(*df=None*, *schema=None*, *metadata=None*)

Bases: *fugue.dataframe.dataframe.LocalUnboundedDataFrame*

DataFrame that wraps native python iterable of arrays. Please read the DataFrame Tutorial to understand the concept

Parameters

- **df** (*Any*) – 2-dimensional array, iterable of arrays, or *DataFrame*
- **schema** (*Any*) – Schema like object
- **metadata** (*Any*) – dict-like object with string keys, default `None`

Examples

```
>>> a = IterableDataFrame([[0,'a'],[1,'b']],"a:int,b:str")
>>> b = IterableDataFrame(a)
```

Note: It's ok to peek the dataframe, it will not affect the iteration, but it's invalid operation to count

alter_columns(*columns*)
Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)
Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

as_array_iterable(*columns=None*, *type_safe=False*)
Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

property empty: bool
Whether this dataframe is empty

property native: triad.utils.iter.EmptyAwareIterable[Any]
Iterable of native python arrays

peek_array()
Peek the first row of the dataframe as array

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Any

rename(*columns*)

Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

fugue.dataframe.pandas_dataframe

class fugue.dataframe.pandas_dataframe.**PandasDataFrame**(*df=None*, *schema=None*, *metadata=None*, *pandas_df_wrapper=False*)

Bases: *fugue.dataframe.dataframe.LocalBoundedDataFrame*

DataFrame that wraps pandas DataFrame. Please also read the DataFrame Tutorial to understand this Fugue concept

Parameters

- **df** (*Any*) – 2-dimensional array, iterable of arrays or pandas DataFrame
- **schema** (*Any*) – Schema like object
- **metadata** (*Any*) – dict-like object with string keys, default None
- **pandas_df_wrapper** (*bool*) – if this is a simple wrapper, default False

Examples

```
>>> PandasDataFrame([[0,'a'],[1,'b']],"a:int,b:str")
>>> PandasDataFrame(schema = "a:int,b:int")  # empty dataframe
>>> PandasDataFrame(pd.DataFrame([[0]],columns=["a"]))
>>> PandasDataFrame(ArrayDataFrame([[0]],"a:int).as_pandas())
```

Note: If pandas_df_wrapper is True, then the constructor will not do any type check otherwise, it will enforce type according to the input schema after the construction

alter_columns(*columns*)

Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)

Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

`as_array_iterable`(*columns=None*, *type_safe=False*)

Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

`as_pandas`()

Convert to pandas DataFrame

Return type pandas.core.frame.DataFrame

`count`()

Get number of rows of this dataframe

Return type int

`property empty: bool`

Whether this dataframe is empty

`head`(*n*, *columns=None*)

Get first n rows of the dataframe as 2-dimensional array

Parameters

- **n** (*int*) – number of rows
- **columns** (*Optional[List[str]]*) – selected columns, defaults to None (all columns)

Returns 2-dimensional array

Return type List[Any]

`property native: pandas.core.frame.DataFrame`

Pandas DataFrame

`peek_array`()

Peek the first row of the dataframe as array

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Any

`rename`(*columns*)

Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

fugue.dataframe.utils

fugue.dataframe.utils.**deserialize_df**(*json_str*, *fs=None*)
Deserialize json string to *LocalBoundedDataFrame*

Parameters

- **json_str** (*str*) – json string containing the base64 data or a file path
- **fs** (*Optional[triad.collections.fs.FileSystem]*) – FileSystem, defaults to None

Raises **ValueError** – if the json string is invalid, not generated from *serialize_df()*

Returns *LocalBoundedDataFrame* if json_str contains a dataframe or None if its valid but contains no data

Return type Optional[*fugue.dataframe.dataframe.LocalBoundedDataFrame*]

fugue.dataframe.utils.**get_join_schemas**(*df1*, *df2*, *how*, *on*)
Get Schema object after joining df1 and df2. If on is not empty, it's mainly for validation purpose.

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – second dataframe
- **how** (*str*) – can accept semi, left_semi, anti, left_anti, inner, left_outer, right_outer, full_outer, cross
- **on** (*Iterable[str]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys.

Returns the pair key schema and schema after join

Return type Tuple[triad.collections.schema.Schema, triad.collections.schema.Schema]

Note: In Fugue, joined schema can always be inferred because it always uses the input dataframes' common keys as the join keys. So you must make sure to *rename()* to input dataframes so they follow this rule.

---

fugue.dataframe.utils.**pickle_df**(*df*)
Pickles a dataframe to bytes array. It firstly converts the dataframe using *to_local_bounded_df()*, and then serialize the underlying data.

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – input DataFrame

Returns pickled binary data

Return type bytes

Note: Be careful to use on large dataframes or non-local, un-materialized dataframes, it can be slow. You should always use *unpickle_df()* to deserialize.

---

fugue.dataframe.utils.**serialize_df**(*df*, *threshold=-1*, *file_path=None*, *fs=None*)
Serialize input dataframe to base64 string or to file if it's larger than threshold

Parameters

- **df** (*Optional[fugue.dataframe.dataframe.DataFrame]*) – input DataFrame
- **threshold** (*int*) – file byte size threshold, defaults to -1
- **file_path** (*Optional[str]*) – file path to store the data (used only if the serialized data is larger than `threshold`), defaults to None
- **fs** (*Optional[triad.collections.fs.FileSystem]*) – `FileSystem`, defaults to None

Raises **InvalidOperationError** – if file is large but `file_path` is not provided

Returns a json string either containing the base64 data or the file path

Return type str

Note: If fs is not provided but it needs to write to disk, then it will use `open_fs()` to try to open the file to write.

---

fugue.dataframe.utils.**to_local_bounded_df**(*df*, *schema=None*, *metadata=None*)

Convert a data structure to *LocalBoundedDataFrame*

Parameters

- **df** (*Any*) – *DataFrame*, pandas DataFramme and list or iterable of arrays
- **schema** (*Optional[Any]*) – Schema like object, defaults to None, it should not be set for *DataFrame* type
- **metadata** (*Optional[Any]*) – dict-like object with string keys, defaults to None

Raises

- **ValueError** – if `df` is *DataFrame* but you set `schema` or `metadata`
- **TypeError** – if `df` is not compatible

Returns the dataframe itself if it's *LocalBoundedDataFrame* else a converted one

Return type *fugue.dataframe.dataframe.LocalBoundedDataFrame*

Examples

```
>>> a = IterableDataFrame([[0,'a'],[1,'b']],"a:int,b:str")
>>> assert isinstance(to_local_bounded_df(a), LocalBoundedDataFrame)
>>> to_local_bounded_df(SparkDataFrame([[0,'a'],[1,'b']],"a:int,b:str"))
```

Note: Compared to *to_local_df()*, this function makes sure the dataframe is also bounded, so *IterableDataFrame* will be converted although it's local.

---

fugue.dataframe.utils.**to_local_df**(*df*, *schema=None*, *metadata=None*)

Convert a data structure to *LocalDataFrame*

Parameters

- **df** (*Any*) – *DataFrame*, pandas DataFramme and list or iterable of arrays
- **schema** (*Optional[Any]*) – Schema like object, defaults to None, it should not be set for *DataFrame* type
- **metadata** (*Optional[Any]*) – dict-like object with string keys, defaults to None

Raises

- **ValueError** – if `df` is *DataFrame* but you set `schema` or `metadata`
- **TypeError** – if `df` is not compatible

Returns the dataframe itself if it's *LocalDataFrame* else a converted one

Return type *fugue.dataframe.dataframe.LocalDataFrame*

Examples

```
>>> a = to_local_df([[0,'a'],[1,'b']],"a:int,b:str")
>>> assert to_local_df(a) is a
>>> to_local_df(SparkDataFrame([[0,'a'],[1,'b']],"a:int,b:str"))
```

fugue.dataframe.utils.**unpickle_df**(*stream*)

Unpickles a dataframe from bytes array.

Parameters **stream** (*bytes*) – binary data

Returns unpickled dataframe

Return type *fugue.dataframe.dataframe.LocalBoundedDataFrame*

Note: The data must be serialized by *pickle_df()* to deserialize.

4.1.4 fugue.execution

fugue.execution.execution_engine

class fugue.execution.execution_engine.**ExecutionEngine**(*conf*)

Bases: abc.ABC

The abstract base class for execution engines. It is the layer that unifies core concepts of distributed computing, and separates the underlying computing frameworks from user's higher level logic.

Please read the ExecutionEngine Tutorial to understand this most important Fugue concept

Parameters **conf** (*Any*) – dict-like config, read this to learn Fugue specific options

aggregate(*df*, *partition_spec*, *agg_cols*, *metadata=None*)

Aggregate on dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the dataframe to aggregate on
- **partition_spec** (*Optional[*fugue.collections.partition.PartitionSpec*]*) – PartitionSpec to specify partition keys
- **agg_cols** (*List[*fugue.column.expressions.ColumnExpr*]*) – aggregation expressions
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None. It's currently not used

Returns the aggregated result as a dataframe

New Since

0.6.0

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
import fugue.column.functions as f

# SELECT MAX(b) AS b FROM df
engine.aggregate(
    df,
    partition_spec=None,
    agg_cols=[f.max(col("b"))])

# SELECT a, MAX(b) AS x FROM df GROUP BY a
engine.aggregate(
    df,
    partition_spec=PartitionSpec(by=["a"]),
    agg_cols=[f.max(col("b")).alias("x")])
```

assign(*df*, *columns*, *metadata=None*)

Update existing columns with new values and add new columns

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the dataframe to set columns
- **columns** (*List[*fugue.column.expressions.ColumnExpr*]*) – column expressions
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None. It's currently not used

Returns the updated dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Tip: This can be used to cast data types, alter column values or add new columns. But you can't use aggregation in columns.

New Since

0.6.0

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
# assume df has schema: a:int,b:str

# add constant column x
engine.assign(df, lit(1,"x"))

# change column b to be a constant integer
engine.assign(df, lit(1,"b"))

# add new x to be a+b
engine.assign(df, (col("a")+col("b")).alias("x"))

# cast column a data type to double
engine.assign(df, col("a").cast(float))
```

abstract broadcast(*df*)

Broadcast the dataframe to all workers for a distributed computing framework

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe

Returns the broadcasted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

comap(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)

Apply a function to each zipped partition on the zipped dataframe.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe, it must be a zipped dataframe (it has to be a dataframe output from *zip()* or *zip_all()*)
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframes.DataFrames*]*, fugue.dataframe.dataframe.LocalDataFrame*]*) – the function to apply on every zipped partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification for processing the zipped zipped dataframe.
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframes.DataFrames*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the comap operation

Note:

- The input of this method must be an output of *zip()* or *zip_all()*
- The `partition_spec` here is NOT related with how you zipped the dataframe and however you set it, will only affect the processing speed, actually the partition keys will be overriden to the zipped dataframe partition keys. You may use it in this way to improve the efficiency: `PartitionSpec(algo="even", num="ROWCOUNT")`, this tells the execution engine to put each zipped partition into a physical partition so it can achieve the best possible load balance.

- If input dataframe has keys, the dataframes you get in `map_func` and `on_init` will have keys, otherwise you will get list-like dataframes
- on_init function will get a DataFrames object that has the same structure, but has all empty dataframes, you can use the schemas but not the data.

See also:

For more details and examples, read Zip & Comap

property compile_conf: `triad.collections.dict.ParamDict`
Compiled time (workflow level) configurations, it is always a superset of conf

Note: Users normally don't need to use this property. It is for internal use.

property conf: `triad.collections.dict.ParamDict`
All configurations of this engine instance.

Note: It can contain more than you providec, for example in *SparkExecutionEngine*, the Spark session can bring in more config, they are all accessible using this property.

convert_yield_dataframe(*df*, *as_local*)
Convert a yield dataframe to a dataframe that can be used after this execution engine stops.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **as_local** (*bool*) – whether yield a local dataframe

Returns another DataFrame that can be used after this execution engine stops

Return type *fugue.dataframe.dataframe.DataFrame*

Note: By default, the output dataframe is the input dataframe. But it should be overridden if when an engine stops and the input dataframe will become invalid.

For example, if you custom a spark engine where you start and stop the spark session in this engine's *start_engine()* and *stop_engine()*, then the spark dataframe will be invalid. So you may consider converting it to a local dataframe so it can still exist after the engine stops.

abstract property default_sql_engine: *fugue.execution.execution_engine.SQLEngine*
Default SQLEngine if user doesn't specify

abstract distinct(*df*, *metadata=None*)
Equivalent to `SELECT DISTINCT * FROM df`

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – dataframe
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns [description]

Return type *DataFrame*

abstract dropna(*df*, *how='any'*, *thresh=None*, *subset=None*, *metadata=None*)
Drop NA recods from dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **how** (*str*) – 'any' or 'all'. 'any' drops rows that contain any nulls. 'all' drops rows that contain all nulls.
- **thresh** (*Optional[int]*) – int, drops rows that have less than thresh non-null values
- **subset** (*Optional[List[str]]*) – list of columns to operate on
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records dropped

Return type *DataFrame*

abstract fillna(*df*, *value*, *subset=None*, *metadata=None*)
Fill NULL, NAN, NAT values in a dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **value** (*Any*) – if scalar, fills all columns with same value. if dictionary, fills NA using the keys as column names and the values as the replacement values.
- **subset** (*Optional[List[str]]*) – list of columns to operate on. ignored if value is a dictionary
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records filled

Return type *DataFrame*

filter(*df*, *condition*, *metadata=None*)
Filter rows by the given condition

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the dataframe to be filtered
- **condition** (fugue.column.expressions.ColumnExpr) – (boolean) column expression
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None. It's currently not used

Returns the filtered dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

New Since

0.6.0

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
import fugue.column.functions as f

engine.filter(df, (col("a")>1) & (col("b")=="x"))
engine.filter(df, f.coalesce(col("a"),col("b"))>1)
```

abstract property fs: triad.collections.fs.FileSystem
File system of this engine instance

abstract intersect(*df1*, *df2*, *distinct=True*, *metadata=None*)
Intersect `df1` and `df2`

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `INTERSECT (== INTERSECT DISTINCT)`, `false` for `INTERSECT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

abstract join(*df1*, *df2*, *how*, *on=[]*, *metadata=None*)
Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **how** (*str*) – can accept `semi`, `left_semi`, `anti`, `left_anti`, `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`
- **on** (*List[str]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys.
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the joined dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Please read *this*

abstract load_df(*path*, *format_hint=None*, *columns=None*, ***kwargs*)
Load dataframe from persistent storage

Parameters

- **path** (*Union[str, List[str]]*) – the path to the dataframe
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **columns** (*Optional[Any]*) – list of columns or a Schema like object, defaults to None
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Returns an engine compatible dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

For more details and examples, read Zip & Comap.

abstract property log: logging.Logger
Logger of this engine instance

abstract map(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)
Apply a function to each partition after you partition the data in a specified way.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframe.LocalDataFrame*], fugue.dataframe.dataframe.LocalDataFrame*]) – the function to apply on every logical partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframe.DataFrame*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the map operation

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing, you must read this to understand what map is used for and how it should work.

---

abstract persist(*df*, *lazy=False*, ***kwargs*)
Force materializing and caching the dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe
- **lazy** (*bool*) – `True`: first usage of the output will trigger persisting to happen; `False` (eager): persist is forced to happend immediately. Default to `False`
- **args** – parameter to pass to the underlying persist implementation
- **kwargs** (*Any*) – parameter to pass to the underlying persist implementation

Returns the persisted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: `persist` can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

---

abstract repartition(*df*, *partition_spec*)

Partition the input dataframe using `partition_spec`.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how you want to partition the dataframe

Returns repartitioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing please read the Partition Tutorial

---

property rpc_server: *fugue.rpc.base.RPCServer*

RPCServer of this execution engine

abstract sample(*df*, *n=None*, *frac=None*, *replace=False*, *seed=None*, *metadata=None*)

Sample dataframe by number of rows or by fraction

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*Optional[int]*) – number of rows to sample, one and only one of `n` and `frac` must be set
- **frac** (*Optional[float]*) – fraction [0,1] to sample, one and only one of `n` and `frac` must be set
- **replace** (*bool*) – whether replacement is allowed. With replacement, there may be duplicated rows in the result, defaults to False
- **seed** (*Optional[int]*) – seed for randomness, defaults to None
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns sampled dataframe

Return type *DataFrame*

abstract save_df(*df*, *path*, *format_hint=None*, *mode='overwrite'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *force_single=False*, ***kwargs*)

Save dataframe to a persistent storage

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **path** (*str*) – output path

- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition_spec** (`fugue.collections.partition.PartitionSpec`) – how to partition the dataframe before saving, defaults to empty
- **force_single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Load & Save.

select(*df*, *cols*, *where=None*, *having=None*, *metadata=None*)

The functional interface for SQL select statement

Parameters

- **df** (`fugue.dataframe.dataframe.DataFrame`) – the dataframe to be operated on
- **cols** (`fugue.column.sql.SelectColumns`) – column expressions
- **where** (*Optional[*`fugue.column.expressions.ColumnExpr`*]*) – `WHERE` condition expression, defaults to None
- **having** (*Optional[*`fugue.column.expressions.ColumnExpr`*]*) – `having` condition expression, defaults to None. It is used when `cols` contains aggregation columns, defaults to None
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None. It's currently not used

Returns the select result as a dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

New Since

0.6.0

Attention: This interface is experimental, it's subjected to change in new versions.

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
import fugue.column.functions as f

# select existed and new columns
engine.select(df, SelectColumns(col("a"),col("b"),lit(1,"another")))
engine.select(df, SelectColumns(col("a"),(col("b")+lit(1)).alias("x")))

# aggregation
```

(continues on next page)

(continued from previous page)

```
# SELECT COUNT(DISTINCT *) AS x FROM df
engine.select(
    df,
    SelectColumns(f.count_distinct(col("*")).alias("x")))

# SELECT a, MAX(b+1) AS x FROM df GROUP BY a
engine.select(
    df,
    SelectColumns(col("a"),f.max(col("b")+lit(1)).alias("x")))

# SELECT a, MAX(b+1) AS x FROM df
#   WHERE b<2 AND a>1
#   GROUP BY a
#   HAVING MAX(b+1)>0
engine.select(
    df,
    SelectColumns(col("a"),f.max(col("b")+lit(1)).alias("x")),
    where=(col("b")<2) & (col("a")>1),
    having=f.max(col("b")+lit(1))>0
)
```

---

set_sql_engine(*engine*)

Set *SQLEngine* for this execution engine. If not set, the default is *default_sql_engine()*

Parameters **engine** (fugue.execution.execution_engine.SQLEngine) – *SQLEngine* instance

Return type None

property sql_engine: *fugue.execution.execution_engine.SQLEngine*

The *SQLEngine* currently used by this execution engine. You should use *set_sql_engine()* to set a new SQLEngine instance. If not set, the default is *default_sql_engine()*

start()

Start this execution engine, do not override. You should customize *start_engine()* if necessary.

Return type None

start_engine()

Custom logic to start the execution engine, defaults to no operation

Return type None

stop()

Stop this execution engine, do not override You should customize *stop_engine()* if necessary.

Return type None

stop_engine()

Custom logic to stop the execution engine, defaults to no operation

Return type None

abstract subtract(*df1*, *df2*, *distinct=True*, *metadata=None*)

df1 - df2

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe

- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `EXCEPT` (== `EXCEPT DISTINCT`), `false` for `EXCEPT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

abstract take(*df*, *n*, *presort*, *na_position='last'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *metadata=None*)

Get the first n rows of a DataFrame per partition. If a presort is defined, use the presort before applying take. presort overrides partition_spec.presort. The Fugue implementation of the presort follows Pandas convention of specifying NULLs first or NULLs last. This is different from the Spark and SQL convention of NULLs as the smallest value.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*int*) – number of rows to return
- **presort** (*str*) – presort expression similar to partition presort
- **na_position** (*str*) – position of null values during the presort. can accept `first` or `last`
- **partition_spec** (fugue.collections.partition.PartitionSpec) – PartitionSpec to apply the take operation
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns n rows of DataFrame per partition

Return type *DataFrame*

abstract to_df(*data*, *schema=None*, *metadata=None*)

Convert a data structure to this engine compatible DataFrame

Parameters

- **data** (*Any*) – *DataFrame*, pandas DataFramme or list or iterable of arrays or others that is supported by certain engine implementation
- **schema** (*Optional[Any]*) – Schema like object, defaults to None
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None

Returns engine compatible dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: There are certain conventions to follow for a new implementation:

- if the input is already in compatible dataframe type, it should return itself
- all other methods in the engine interface should take arbitrary dataframes and call this method to convert before doing anything

abstract union(*df1*, *df2*, *distinct=True*, *metadata=None*)
Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `UNION` (== `UNION DISTINCT`), `false` for `UNION ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

zip(*df1*, *df2*, *how='inner'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *temp_path=None*, *to_file_threshold=-1*, *df1_name=None*, *df2_name=None*)
Partition the two dataframes in the same way with `partition_spec` and zip the partitions together on the partition keys.

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **how** (*str*) – can accept `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`, defaults to `inner`
- **partition_spec** (PartitionSpec, *optional*) – partition spec to partition each dataframe, defaults to empty.
- **temp_path** (*Optional[str]*) – file path to store the data (used only if the serialized data is larger than `to_file_threshold`), defaults to None
- **to_file_threshold** (*Any*) – file byte size threshold, defaults to -1
- **df1_name** (*Optional[str]*) – df1's name in the zipped dataframe, defaults to None
- **df2_name** (*Optional[str]*) – df2's name in the zipped dataframe, defaults to None

Returns a zipped dataframe, the metadata of the dataframe will indicate it's zipped

Note:

- Different from join, `df1` and `df2` can have common columns that you will not use as partition keys.
- If `on` is not specified it will also use the common columns of the two dataframes (if it's not a cross zip)
- For non-cross zip, the two dataframes must have common columns, or error will be thrown

See also:

For more details and examples, read Zip & Comap.

zip_all(*dfs*, *how='inner'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *temp_path=None*, *to_file_threshold=-1*)

Zip multiple dataframes together with given partition specifications.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – DataFrames like object
- **how** (*str*) – can accept `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`, defaults to `inner`
- **partition_spec** (fugue.collections.partition.PartitionSpec) – Partition like object, defaults to empty.
- **temp_path** (*Optional[str]*) – file path to store the data (used only if the serialized data is larger than `to_file_threshold`), defaults to None
- **to_file_threshold** (*Any*) – file byte size threshold, defaults to -1

Returns a zipped dataframe, the metadata of the dataframe will indicated it's zipped

Return type *fugue.dataframe.dataframe.DataFrame*

Note:

- Please also read *zip()*
- If `dfs` is dict like, the zipped dataframe will be dict like, If `dfs` is list like, the zipped dataframe will be list like
- It's fine to contain only one dataframe in `dfs`

See also:

For more details and examples, read Zip & Comap

class fugue.execution.execution_engine.**SQLEngine**(*execution_engine*)

Bases: `abc.ABC`

The abstract base class for different SQL execution implementations. Please read this to understand the concept

Parameters **execution_engine** (ExecutionEngine) – the execution engine this sql engine will run on

Return type None

property execution_engine: *fugue.execution.execution_engine.ExecutionEngine*

the execution engine this sql engine will run on

abstract select(*dfs*, *statement*)

Execute select statement on the sql engine.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **statement** (*str*) – the `SELECT` statement using the `dfs` keys as tables

Returns result of the `SELECT` statement

Return type *fugue.dataframe.dataframe.DataFrame*

Examples

```
>>> dfs = DataFrames(a=df1, b=df2)
>>> sql_engine.select(dfs, "SELECT * FROM a UNION SELECT * FROM b")
```

Note: There can be tables that is not in `dfs`. For example you want to select from hive without input DataFrames:

```
>>> sql_engine.select(DataFrames(), "SELECT * FROM hive.a.table")
```

fugue.execution.factory

fugue.execution.factory.**make_execution_engine**(*engine=None*, *conf=None*, ***kwargs*)

Create *ExecutionEngine* with specified `engine`

Parameters

- **engine** (*Optional[Any]*) – it can be empty string or null (use the default execution engine), a string (use the registered execution engine), an *ExecutionEngine* type, or the *ExecutionEngine* instance , or a tuple of two values where the first value represents execution engine and the second value represents the sql engine (you can use `None` for either of them to use the default one), defaults to None
- **conf** (*Optional[Any]*) – Parameters like object, defaults to None
- **kwargs** (*Any*) – additional parameters to initialize the execution engine

Returns the *ExecutionEngine* instance

Return type *fugue.execution.execution_engine.ExecutionEngine*

Examples

```
register_default_execution_engine(lambda conf: E1(conf))
register_execution_engine("e2", lambda conf, **kwargs: E2(conf, **kwargs))

register_sql_engine("s", lambda conf: S2(conf))

# E1 + E1.default_sql_engine
make_execution_engine()

# E2 + E2.default_sql_engine
make_execution_engine(e2)

# E1 + S2
make_execution_engine((None, "s"))

# E2(conf, a=1, b=2) + S2
make_execution_engine(("e2", "s"), conf, a=1, b=2)

# SparkExecutionEngine + SparkSQLEngine
make_execution_engine(SparkExecutionEngine)
make_execution_engine(SparkExecutionEngine(spark_session, conf))
```

(continues on next page)

(continued from previous page)

```
# SparkExecutionEngine + S2
make_execution_engine((SparkExecutionEngine, "s"))
```

fugue.execution.factory.**make_sql_engine**(*engine=None*, *execution_engine=None*, ***kwargs*)

Create *SQLEngine* with specified `engine`

Parameters

- **engine** (*Optional[Any]*) – it can be empty string or null (use the default SQL engine), a string (use the registered SQL engine), an *SQLEngine* type, or the *SQLEngine* instance (you can use `None` to use the default one), defaults to None
- **execution_engine** (*Optional[fugue.execution.execution_engine.ExecutionEngine]*) – the *ExecutionEngine* instance to create the *SQLEngine*. Normally you should always provide this value.
- **kwargs** (*Any*) – additional parameters to initialize the sql engine

Returns the *SQLEngine* instance

Return type *fugue.execution.execution_engine.SQLEngine*

Note: For users, you normally don't need to call this function directly. Use `make_execution_engine` instead

Examples

```
register_default_sql_engine(lambda conf: S1(conf))
register_sql_engine("s2", lambda conf: S2(conf))

engine = NativeExecutionEngine()

# S1(engine)
make_sql_engine(None, engine)

# S1(engine, a=1)
make_sql_engine(None, engine, a=1)

# S2(engine)
make_sql_engine("s2", engine)

# SqliteEngine(engine)
make_sql_engine(SqliteEngine)
```

fugue.execution.factory.**register_default_execution_engine**(*func*, *on_dup='overwrite'*)

Register *ExecutionEngine* as the default engine.

Parameters

- **func** (*Callable*) – a callable taking Parameters like object and `**kwargs` and returning an *ExecutionEngine* instance

- **on_dup** – action on duplicated `name`. It can be "overwrite", "ignore" (not overwriting), defaults to "overwrite".

Return type None

Examples

```
# create a new engine with name my (overwrites if existed)
register_default_execution_engine(lambda conf: MyExecutionEngine(conf))

# the following examples will use MyExecutionEngine

# 0
make_execution_engine()
make_execution_engine(None, {"myconfig":"value})

# 1
with FugueWorkflow() as dag:
    dag.create([[0]],"a:int").show()

# 2
dag = FugueWorkflow()
dag.create([[0]],"a:int").show()
dag.run(None, {"myconfig":"value})

# 3
fsql('''
CREATE [[0]] SCHEMA a:int
PRINT
''').run("", {"myconfig":"value})
```

fugue.execution.factory.**register_default_sql_engine**(*func*, *on_dup='overwrite'*)

Register *SQLEngine* as the default engine

Parameters

- **func** (*Callable*) – a callable taking *ExecutionEngine* and ****kwargs** and returning a *SQLEngine* instance
- **on_dup** – action on duplicated `name`. It can be "overwrite", "ignore" (not overwriting) or "throw" (throw exception), defaults to "overwrite".

Raises **KeyError** – if `on_dup` is `throw` and the `name` already exists

Return type None

Note: You should be careful to use this function, because when you set a custom SQL engine as default, all execution engines you create will use this SQL engine unless you are explicit. For example if you set the default SQL engine to be a Spark specific one, then if you start a NativeExecutionEngine, it will try to use it and will throw exceptions.

So it's always a better idea to use `register_sql_engine` instead

Examples

```
# create a new engine with name my (overwrites if existed)
register_default_sql_engine(lambda engine: MySQLEngine(engine))

# create NativeExecutionEngine with MySQLEngine as the default
make_execution_engine()

# create SparkExecutionEngine with MySQLEngine instead of SparkSQLEngine
make_execution_engine("spark")

# NativeExecutionEngine with MySQLEngine
with FugueWorkflow() as dag:
    dag.create([[0]],"a:int").show()
```

---

fugue.execution.factory.**register_execution_engine**(*name_or_type*, *func*, *on_dup='overwrite'*)

Register *ExecutionEngine* with a given name.

Parameters

- **name_or_type** (*Union[str, Type]*) – alias of the execution engine, or type of an object that can be converted to an execution engine
- **func** (*Callable*) – a callable taking Parameters like object and **kwargs and returning an *ExecutionEngine* instance
- **on_dup** – action on duplicated name. It can be "overwrite", "ignore" (not overwriting), defaults to "overwrite".

Return type None

Examples

Alias registration examples:

```
# create a new engine with name my (overwrites if existed)
register_execution_engine("my", lambda conf: MyExecutionEngine(conf))

# 0
make_execution_engine("my")
make_execution_engine("my", {"myconfig":"value})

# 1
with FugueWorkflow("my") as dag:
    dag.create([[0]],"a:int").show()

# 2
dag = FugueWorkflow()
dag.create([[0]],"a:int").show()
dag.run("my", {"myconfig":"value})

# 3
fsql('''
CREATE [[0]] SCHEMA a:int
PRINT
''').run("my")
```

Type registration examples:

```
from pyspark.sql import SparkSession
from fugue_spark import SparkExecutionEngine
from fugue_sql import fsql

register_execution_engine(
    SparkSession,
    lambda session, conf: SparkExecutionEngine(session, conf))

spark_session = SparkSession.builder.getOrCreate()

fsql('''
CREATE [[0]] SCHEMA a:int
PRINT
''').run(spark_session)
```

---

fugue.execution.factory.**register_sql_engine**(*name*, *func*, *on_dup='overwrite'*)

Register *SQLEngine* with a given name.

Parameters

- **name** (*str*) – name of the SQL engine
- **func** (*Callable*) – a callable taking *ExecutionEngine* and **kwargs and returning a *SQLEngine* instance
- **on_dup** – action on duplicated name. It can be "overwrite", "ignore" (not overwriting), defaults to "overwrite".

Return type None

Examples

```
# create a new engine with name my (overwrites if existed)
register_sql_engine("mysql", lambda engine: MySQLEngine(engine))

# create execution engine with MySQLEngine as the default
make_execution_engine(("", "mysql"))

# create DaskExecutionEngine with MySQLEngine as the default
make_execution_engine(("dask", "mysql"))

# default execution engine + MySQLEngine
with FugueWorkflow(("","mysql")) as dag:
    dag.create([[0]],"a:int").show()
```

---

fugue.execution.native_execution_engine

class fugue.execution.native_execution_engine.**NativeExecutionEngine**(*conf=None*)
Bases: *fugue.execution.execution_engine.ExecutionEngine*

The execution engine based on native python and pandas. This execution engine is mainly for prototyping and unit tests.

Please read the ExecutionEngine Tutorial to understand this important Fugue concept

Parameters **conf** (*Any*) – Parameters like object, read the Fugue Configuration Tutorial to learn Fugue specific options

broadcast(*df*)
Broadcast the dataframe to all workers for a distributed computing framework

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe

Returns the broadcasted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

property default_sql_engine: *fugue.execution.execution_engine.SQLEngine*
Default SQLEngine if user doesn't specify

distinct(*df*, *metadata=None*)
Equivalent to `SELECT DISTINCT * FROM df`

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – dataframe
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns [description]

Return type *DataFrame*

dropna(*df*, *how='any'*, *thresh=None*, *subset=None*, *metadata=None*)
Drop NA recods from dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **how** (*str*) – 'any' or 'all'. 'any' drops rows that contain any nulls. 'all' drops rows that contain all nulls.
- **thresh** (*Optional[int]*) – int, drops rows that have less than thresh non-null values
- **subset** (*Optional[List[str]]*) – list of columns to operate on
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records dropped

Return type *DataFrame*

fillna(*df*, *value*, *subset=None*, *metadata=None*)
Fill `NULL`, `NAN`, `NAT` values in a dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame

- **value** (*Any*) – if scalar, fills all columns with same value. if dictionary, fills NA using the keys as column names and the values as the replacement values.
- **subset** (*Optional[List[str]]*) – list of columns to operate on. ignored if value is a dictionary
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records filled

Return type *DataFrame*

property fs: triad.collections.fs.FileSystem
File system of this engine instance

intersect(*df1*, *df2*, *distinct=True*, *metadata=None*)
Intersect `df1` and `df2`

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `INTERSECT (== INTERSECT DISTINCT)`, `false` for `INTERSECT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

join(*df1*, *df2*, *how*, *on=[]*, *metadata=None*)
Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **how** (*str*) – can accept `semi`, `left_semi`, `anti`, `left_anti`, `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`
- **on** (*List[str]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys.
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the joined dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Please read *this*

load_df(*path*, *format_hint=None*, *columns=None*, ***kwargs*)
Load dataframe from persistent storage

Parameters

- **path** (*Union[str, List[str]]*) – the path to the dataframe
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **columns** (*Optional[Any]*) – list of columns or a Schema like object, defaults to None
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Returns an engine compatible dataframe

Return type *fugue.dataframe.dataframe.LocalBoundedDataFrame*

For more details and examples, read Zip & Comap.

property log: logging.Logger
Logger of this engine instance

map(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)
Apply a function to each partition after you partition the data in a specified way.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframe.LocalDataFrame*], *fugue.dataframe.dataframe.LocalDataFrame*]) – the function to apply on every logical partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframe.DataFrame*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the map operation

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing, you must read this to understand what map is used for and how it should work.

---

persist(*df*, *lazy=False*, ***kwargs*)
Force materializing and caching the dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe
- **lazy** (*bool*) – `True`: first usage of the output will trigger persisting to happen; `False` (eager): persist is forced to happend immediately. Default to `False`

- **args** – parameter to pass to the underlying persist implementation
- **kwargs** (*Any*) – parameter to pass to the underlying persist implementation

Returns the persisted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: `persist` can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

---

property pl_utils: qpd_pandas.engine.PandasUtils
Pandas-like dataframe utils

repartition(*df*, *partition_spec*)
Partition the input dataframe using `partition_spec`.

Parameters

- **df** (`fugue.dataframe.dataframe.DataFrame`) – input dataframe
- **partition_spec** (`fugue.collections.partition.PartitionSpec`) – how you want to partition the dataframe

Returns repartitioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing please read the Partition Tutorial

---

sample(*df*, *n=None*, *frac=None*, *replace=False*, *seed=None*, *metadata=None*)
Sample dataframe by number of rows or by fraction

Parameters

- **df** (`fugue.dataframe.dataframe.DataFrame`) – DataFrame
- **n** (*Optional[int]*) – number of rows to sample, one and only one of `n` and `frac` must be set
- **frac** (*Optional[float]*) – fraction [0,1] to sample, one and only one of `n` and `frac` must be set
- **replace** (*bool*) – whether replacement is allowed. With replacement, there may be duplicated rows in the result, defaults to False
- **seed** (*Optional[int]*) – seed for randomness, defaults to None
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns sampled dataframe

Return type *DataFrame*

save_df(*df*, *path*, *format_hint=None*, *mode='overwrite'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *force_single=False*, ***kwargs*)
Save dataframe to a persistent storage

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **path** (*str*) – output path
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how to partition the dataframe before saving, defaults to empty
- **force_single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Load & Save.

subtract(*df1*, *df2*, *distinct=True*, *metadata=None*)

`df1 - df2`

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `EXCEPT` (== `EXCEPT DISTINCT`), `false` for `EXCEPT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

take(*df*, *n*, *presort*, *na_position='last'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *metadata=None*)

Get the first n rows of a DataFrame per partition. If a presort is defined, use the presort before applying take. presort overrides partition_spec.presort. The Fugue implementation of the presort follows Pandas convention of specifying NULLs first or NULLs last. This is different from the Spark and SQL convention of NULLs as the smallest value.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*int*) – number of rows to return
- **presort** (*str*) – presort expression similar to partition presort
- **na_position** (*str*) – position of null values during the presort. can accept `first` or `last`
- **partition_spec** (fugue.collections.partition.PartitionSpec) – PartitionSpec to apply the take operation
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns n rows of DataFrame per partition

Return type *DataFrame*

to_df(*df*, *schema=None*, *metadata=None*)
Convert a data structure to this engine compatible DataFrame

Parameters

- **data** – *DataFrame*, pandas DataFramme or list or iterable of arrays or others that is supported by certain engine implementation
- **schema** (*Optional[Any]*) – Schema like object, defaults to None
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None
- **df** (*Any*) –

Returns engine compatible dataframe

Return type *fugue.dataframe.dataframe.LocalBoundedDataFrame*

Note: There are certain conventions to follow for a new implementation:

- if the input is already in compatible dataframe type, it should return itself
- all other methods in the engine interface should take arbitrary dataframes and call this method to convert before doing anything

---

union(*df1*, *df2*, *distinct=True*, *metadata=None*)
Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `UNION` (== `UNION DISTINCT`), `false` for `UNION ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

class fugue.execution.native_execution_engine.**QPDPandasEngine**(*execution_engine*)
Bases: *fugue.execution.execution_engine.SQLEngine*

QPD execution implementation.

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) – the execution engine this sql engine will run on

select(*dfs*, *statement*)
Execute select statement on the sql engine.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys

- **statement** (*str*) – the SELECT statement using the dfs keys as tables

Returns result of the SELECT statement

Return type *fugue.dataframe.dataframe.DataFrame*

Examples

```
>>> dfs = DataFrames(a=df1, b=df2)
>>> sql_engine.select(dfs, "SELECT * FROM a UNION SELECT * FROM b")
```

Note: There can be tables that is not in dfs. For example you want to select from hive without input DataFrames:

```
>>> sql_engine.select(DataFrames(), "SELECT * FROM hive.a.table")
```

class fugue.execution.native_execution_engine.**SqliteEngine**(*execution_engine*)

Bases: *fugue.execution.execution_engine.SQLEngine*

Sqlite execution implementation.

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) – the execution engine this sql engine will run on

select(*dfs*, *statement*)

Execute select statement on the sql engine.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **statement** (*str*) – the SELECT statement using the dfs keys as tables

Returns result of the SELECT statement

Return type *fugue.dataframe.dataframe.DataFrame*

Examples

```
>>> dfs = DataFrames(a=df1, b=df2)
>>> sql_engine.select(dfs, "SELECT * FROM a UNION SELECT * FROM b")
```

Note: There can be tables that is not in dfs. For example you want to select from hive without input DataFrames:

```
>>> sql_engine.select(DataFrames(), "SELECT * FROM hive.a.table")
```

4.1.5 fugue.extensions

fugue.extensions.creator

fugue.extensions.creator.convert

fugue.extensions.creator.convert.**creator**(*schema=None*)

Decorator for creators

Please read Creator Tutorial

Parameters **schema** (*Optional[Any]*) –

Return type Callable[[Any], fugue.extensions.creator.convert._FuncAsCreator]

fugue.extensions.creator.convert.**register_creator**(*alias*, *obj*, *on_dup='overwrite'*)

Register creator with an alias.

Parameters

- **alias** (*str*) – alias of the creator
- **obj** (*Any*) – the object that can be converted to *Creator*
- **on_dup** (*str*) – action on duplicated `alias`. It can be "overwrite", "ignore" (not overwriting) or "throw" (throw exception), defaults to "overwrite".

Return type None

Tip: Registering an extension with an alias is particularly useful for projects such as libraries. This is because by using alias, users don't have to import the specific extension, or provide the full path of the extension. It can make user's code less dependent and easy to understand.

New Since

0.6.0

See also:

Please read Creator Tutorial

Examples

Here is an example how you setup your project so your users can benefit from this feature. Assume your project name is `pn`

The creator implementation in file `pn/pn/creators.py`

```
import pandas import pd

def my_creator() -> pd.DataFrame:
    return pd.DataFrame()
```

Then in `pn/pn/__init__.py`

```
from .creators import my_creator
from fugue import register_creator

def register_extensions():
    register_creator("mc", my_creator)
    # ... register more extensions

register_extensions()
```

In users code:

```
import pn  # register_extensions will be called
from fugue import FugueWorkflow

with FugueWorkflow() as dag:
    dag.create("mc").show()  # use my_creator by alias
```

---

fugue.extensions.creator.creator

class fugue.extensions.creator.creator.**Creator**

Bases: *fugue.extensions.context.ExtensionContext*, abc.ABC

The interface is to generate single DataFrame from *params*. For example reading data from file should be a type of Creator. Creator is task level extension, running on driver, and execution engine aware.

To implement this class, you should not have `__init__`, please directly implement the interface functions.

Note: Before implementing this class, do you really need to implement this interface? Do you know the interfaceless feature of Fugue? Implementing Creator is commonly unnecessary. You can choose the interfaceless approach which may decouple your code from Fugue.

See also:

Please read Creator Tutorial

abstract create()

Create DataFrame on driver side

Note:

- It runs on driver side
- The output dataframe is not necessarily local, for example a SparkDataFrame
- It is engine aware, you can put platform dependent code in it (for example native pyspark code) but by doing so your code may not be portable. If you only use the functions of the general ExecutionEngine interface, it's still portable.

Returns result dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

fugue.extensions.outputter

fugue.extensions.outputter.convert

fugue.extensions.outputter.convert.**outputter**(***validation_rules*)
Decorator for outputters

Please read Outputter Tutorial

Parameters **validation_rules** (*Any*) –

Return type Callable[[Any], fugue.extensions.outputter.convert._FuncAsOutputter]

fugue.extensions.outputter.convert.**register_outputter**(*alias*, *obj*, *on_dup='overwrite'*)
Register outputter with an alias.

Parameters

- **alias** (*str*) – alias of the processor
- **obj** (*Any*) – the object that can be converted to *Outputter*
- **on_dup** (*str*) – action on duplicated `alias`. It can be "overwrite", "ignore" (not overwriting) or "throw" (throw exception), defaults to "overwrite".

Return type None

Tip: Registering an extension with an alias is particularly useful for projects such as libraries. This is because by using alias, users don't have to import the specific extension, or provide the full path of the extension. It can make user's code less dependent and easy to understand.

New Since

0.6.0

See also:

Please read Outputter Tutorial

Examples

Here is an example how you setup your project so your users can benefit from this feature. Assume your project name is `pn`

The processor implementation in file `pn/pn/outputters.py`

```
from fugue import DataFrame

def my_outputter(df:DataFrame) -> None:
    print(df)
```

Then in `pn/pn/__init__.py`

```
from .outputters import my_outputter
from fugue import register_outputter

def register_extensions():
```

(continues on next page)

(continued from previous page)

```
    register_outputter("mo", my_outputter)
    # ... register more extensions

register_extensions()
```

In users code:

```
import pn  # register_extensions will be called
from fugue import FugueWorkflow

with FugueWorkflow() as dag:
    # use my_outputter by alias
    dag.df([[0]],"a:int").output("mo")
```

---

fugue.extensions.outputter.outputter

class fugue.extensions.outputter.outputter.**Outputter**

Bases: *fugue.extensions.context.ExtensionContext*, abc.ABC

The interface to process one or multiple incoming dataframes without returning anything. For example printing or saving dataframes should be a type of Outputter. Outputter is task level extension, running on driver, and execution engine aware.

To implement this class, you should not have `__init__`, please directly implement the interface functions.

Note: Before implementing this class, do you really need to implement this interface? Do you know the interfaceless feature of Fugue? Implementing Outputter is commonly unnecessary. You can choose the interfaceless approach which may decouple your code from Fugue.

See also:

Please read Outputter Tutorial

abstract process(*dfs*)

Process the collection of dataframes on driver side

Note:

- It runs on driver side
- The dataframes are not necessarily local, for example a SparkDataFrame
- It is engine aware, you can put platform dependent code in it (for example native pyspark code) but by doing so your code may not be portable. If you only use the functions of the general ExecutionEngine, it's still portable.

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) – dataframe collection to process

Return type None

fugue.extensions.processor

fugue.extensions.processor.convert

fugue.extensions.processor.convert.**processor**(*schema=None*, ***validation_rules*)

Decorator for processors

Please read Processor Tutorial

Parameters

- **schema** (*Optional[Any]*) –
- **validation_rules** (*Any*) –

Return type Callable[[Any], fugue.extensions.processor.convert._FuncAsProcessor]

fugue.extensions.processor.convert.**register_processor**(*alias*, *obj*, *on_dup='overwrite'*)

Register processor with an alias.

Parameters

- **alias** (*str*) – alias of the processor
- **obj** (*Any*) – the object that can be converted to *Processor*
- **on_dup** (*str*) – action on duplicated `alias`. It can be "overwrite", "ignore" (not overwriting) or "throw" (throw exception), defaults to "overwrite".

Return type None

Tip: Registering an extension with an alias is particularly useful for projects such as libraries. This is because by using alias, users don't have to import the specific extension, or provide the full path of the extension. It can make user's code less dependent and easy to understand.

New Since

0.6.0

See also:

Please read Processor Tutorial

Examples

Here is an example how you setup your project so your users can benefit from this feature. Assume your project name is `pn`

The processor implementation in file `pn/pn/processors.py`

```
from fugue import DataFrame

def my_processor(df:DataFrame) -> DataFrame:
    return df
```

Then in `pn/pn/__init__.py`

```
from .processors import my_processor
from fugue import register_processor

def register_extensions():
    register_processor("mp", my_processor)
    # ... register more extensions

register_extensions()
```

In users code:

```
import pn  # register_extensions will be called
from fugue import FugueWorkflow

with FugueWorkflow() as dag:
    # use my_processor by alias
    dag.df([[0]],"a:int").process("mp").show()
```

---

fugue.extensions.processor.processor

class fugue.extensions.processor.processor.**Processor**

Bases: *fugue.extensions.context.ExtensionContext*, abc.ABC

The interface to process one or multiple incoming dataframes and return one DataFrame. For example dropping a column of df should be a type of Processor. Processor is task level extension, running on driver, and execution engine aware.

To implement this class, you should not have `__init__`, please directly implement the interface functions.

Note: Before implementing this class, do you really need to implement this interface? Do you know the interfaceless feature of Fugue? Implementing Processor is commonly unnecessary. You can choose the interfaceless approach which may decouple your code from Fugue.

See also:

Please read Processor Tutorial

abstract process(*dfs*)

Process the collection of dataframes on driver side

Note:

- It runs on driver side
- The dataframes are not necessarily local, for example a SparkDataFrame
- It is engine aware, you can put platform dependent code in it (for example native pyspark code) but by doing so your code may not be portable. If you only use the functions of the general ExecutionEngine, it's still portable.

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) – dataframe collection to process

Returns the result dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

fugue.extensions.transformer

fugue.extensions.transformer.constants

fugue.extensions.transformer.convert

fugue.extensions.transformer.convert.**cotransformer**(*schema*, ***validation_rules*)
Decorator for cotransformers

Please read the CoTransformer Tutorial

Parameters

- **schema** (*Any*) –
- **validation_rules** (*Any*) –

Return type Callable[[Any], fugue.extensions.transformer.convert._FuncAsCoTransformer]

fugue.extensions.transformer.convert.**output_cotransformer**(***validation_rules*)
Decorator for cotransformers

Please read the CoTransformer Tutorial

Parameters **validation_rules** (*Any*) –

Return type Callable[[Any], fugue.extensions.transformer.convert._FuncAsCoTransformer]

fugue.extensions.transformer.convert.**output_transformer**(***validation_rules*)
Decorator for transformers

Please read the Transformer Tutorial

Parameters **validation_rules** (*Any*) –

Return type Callable[[Any], fugue.extensions.transformer.convert._FuncAsTransformer]

fugue.extensions.transformer.convert.**register_output_transformer**(*alias*, *obj*, *on_dup='overwrite'*)
Register output transformer with an alias.

Parameters

- **alias** (*str*) – alias of the transformer
- **obj** (*Any*) – the object that can be converted to *OutputTransformer* or *OutputCoTransformer*
- **on_dup** (*str*) – action on duplicated `alias`. It can be "overwrite", "ignore" (not overwriting) or "throw" (throw exception), defaults to "overwrite".

Return type None

Tip: Registering an extension with an alias is particularly useful for projects such as libraries. This is because by using alias, users don't have to import the specific extension, or provide the full path of the extension. It can make user's code less dependent and easy to understand.

New Since

0.6.0

See also:

Please read the Transformer Tutorial

Examples

Here is an example how you setup your project so your users can benefit from this feature. Assume your project name is pn

The transformer implementation in file pn/pn/transformers.py

```
import pandas as pd

def my_transformer(df:pd.DataFrame) -> None:
    df.to_parquet("<unique_path>")
```

Then in pn/pn/__init__.py

```
from .transformers import my_transformer
from fugue import register_transformer

def register_extensions():
    register_transformer("mt", my_transformer)
    # ... register more extensions

register_extensions()
```

In users code:

```
import pn  # register_extensions will be called
from fugue import FugueWorkflow

with FugueWorkflow() as dag:
    # use my_transformer by alias
    dag.df([[0]],"a:int").out_transform("mt")
```

fugue.extensions.transformer.convert.**register_transformer**(*alias*, *obj*, *on_dup='overwrite'*)

Register transformer with an alias.

Parameters

- **alias** (*str*) – alias of the transformer
- **obj** (*Any*) – the object that can be converted to *Transformer* or *CoTransformer*
- **on_dup** (*str*) – action on duplicated `alias`. It can be "overwrite", "ignore" (not overwriting) or "throw" (throw exception), defaults to "overwrite".

Return type None

Tip: Registering an extension with an alias is particularly useful for projects such as libraries. This is because by using alias, users don't have to import the specific extension, or provide the full path of the extension. It can make user's code less dependent and easy to understand.

New Since

0.6.0

See also:

Please read the Transformer Tutorial

Examples

Here is an example how you setup your project so your users can benefit from this feature. Assume your project name is pn

The transformer implementation in file pn/pn/transformers.py

```
import pandas as pd

# schema: *
def my_transformer(df:pd.DataFrame) -> pd.DataFrame:
    return df
```

Then in pn/pn/__init__.py

```
from .transformers import my_transformer
from fugue import register_transformer

def register_extensions():
    register_transformer("mt", my_transformer)
    # ... register more extensions

register_extensions()
```

In users code:

```
import pn  # register_extensions will be called
from fugue import FugueWorkflow

with FugueWorkflow() as dag:
    # use my_transformer by alias
    dag.df([[0]],"a:int").transform("mt").show()
```

fugue.extensions.transformer.convert.**transformer**(*schema*, ***validation_rules*)

Decorator for transformers

Please read the Transformer Tutorial

Parameters

- **schema** (*Any*) –
- **validation_rules** (*Any*) –

Return type Callable[[Any], fugue.extensions.transformer.convert._FuncAsTransformer]

fugue.extensions.transformer.transformer

class fugue.extensions.transformer.transformer.**CoTransformer**

Bases: *fugue.extensions.context.ExtensionContext*

The interface to process logical partitions of a zipped dataframe. A dataframe such as SparkDataFrame can be distributed. But this interface is about local process, scalability and throughput is not a concern of CoTransformer.

To implement this class, you should not have `__init__`, please directly implement the interface functions.

Note: Before implementing this class, do you really need to implement this interface? Do you know the interfaceless feature of Fugue? Commonly, if you don't need to implement *on_init()*, you can choose the interfaceless approach which may decouple your code from Fugue.

It's important to understand Zip & Comap, and please also read the CoTransformer Tutorial

Due to similar issue on spark pickling ABC objects. This class is not ABC. If you encounter the similar issue, possible solution at

---

get_output_schema(*dfs*)

Generate the output schema on the driver side.

Note:

- This is running on driver
- Currently, `dfs` is a collection of empty dataframes with the same structure and schemas
- Normally, you should not consume this dataframe in this step, and you s hould only use its schema and metadata
- You can access all properties except for *cursor()*

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) – the collection of dataframes you are going to transform. They are empty dataframes with the same structure and schemas

Returns Schema like object, should not be None or empty

Return type Any

on_init(*dfs*)

Callback for initializing physical partition that contains one or multiple logical partitions. You may put expensive initialization logic here so you will not have to repeat that in *transform()*

Note:

- This call can be on a random machine (depending on the ExecutionEngine you use), you should get the context from the properties of this class
- You can get physical partition no (if available from the execution egnine) from *cursor()*
- Currently, `dfs` is a collection of empty dataframes with the same structure and schemas

---

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of empty dataframes with the same structure and schemas

Return type None

`transform`(*dfs*)
The transformation logic from a collection of dataframes (with the same partition keys) to a local dataframe.

Note:

- This call can be on a random machine (depending on the ExecutionEngine you use), you should get the *context* from the properties of this class

---

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes with the same partition keys

Returns transformed dataframe

Return type *fugue.dataframe.dataframe.LocalDataFrame*

`class` fugue.extensions.transformer.transformer.**OutputCoTransformer**
Bases: *fugue.extensions.transformer.transformer.CoTransformer*

`get_output_schema`(*dfs*)
Generate the output schema on the driver side.

Note:

- This is running on driver
- Currently, `dfs` is a collection of empty dataframes with the same structure and schemas
- Normally, you should not consume this dataframe in this step, and you s hould only use its schema and metadata
- You can access all properties except for *cursor()*

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) – the collection of dataframes you are going to transform. They are empty dataframes with the same structure and schemas

Returns Schema like object, should not be None or empty

Return type Any

`process`(*dfs*)

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) –

Return type None

`transform`(*dfs*)
The transformation logic from a collection of dataframes (with the same partition keys) to a local dataframe.

Note:

- This call can be on a random machine (depending on the ExecutionEngine you use), you should get the *context* from the properties of this class

---

Parameters **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes with the same partition keys

Returns transformed dataframe

Return type *fugue.dataframe.dataframe.LocalDataFrame*

class fugue.extensions.transformer.transformer.**OutputTransformer**

Bases: *fugue.extensions.transformer.transformer.Transformer*

get_output_schema(*df*)

Generate the output schema on the driver side.

Note:

- This is running on driver
- This is the only function in this interface that is facing the entire DataFrame that is not necessarily local, for example a SparkDataFrame
- Normally, you should not consume this dataframe in this step, and you s hould only use its schema and metadata
- You can access all properties except for *cursor()*

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the entire dataframe you are going to transform.

Returns Schema like object, should not be None or empty

Return type Any

process(*df*)

Parameters **df** (fugue.dataframe.dataframe.LocalDataFrame) –

Return type None

transform(*df*)

The transformation logic from one local dataframe to another local dataframe.

Note:

- This function operates on logical partition level
- This call can be on a random machine (depending on the ExecutionEngine you use), you should get the *context* from the properties of this class
- The input dataframe may be unbounded, but must be empty aware. It's safe to consume it for ONLY ONCE
- The input dataframe is never empty. Empty dataframes are skipped

---

Parameters **df** (fugue.dataframe.dataframe.LocalDataFrame) – one logical partition to transform on

Returns transformed dataframe

Return type *fugue.dataframe.dataframe.LocalDataFrame*

class fugue.extensions.transformer.transformer.**Transformer**

Bases: *fugue.extensions.context.ExtensionContext*

The interface to process logical partitions of a dataframe. A dataframe such as SparkDataFrame can be distributed. But this interface is about local process, scalability and throughput is not a concern of Transformer.

To implement this class, you should not have `__init__`, please directly implement the interface functions.

Note: Before implementing this class, do you really need to implement this interface? Do you know the interfaceless feature of Fugue? Commonly, if you don't need to implement *on_init()*, you can choose the interfaceless approach which may decouple your code from Fugue.

It's important to understand the Partition Tutorial, and please also read the Transformer Tutorial

Due to similar issue on spark pickling ABC objects. This class is not ABC. If you encounter the similar issue, possible solution at

---

get_output_schema(*df*)

Generate the output schema on the driver side.

Note:

- This is running on driver
- This is the only function in this interface that is facing the entire DataFrame that is not necessarily local, for example a SparkDataFrame
- Normally, you should not consume this dataframe in this step, and you s hould only use its schema and metadata
- You can access all properties except for *cursor()*

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the entire dataframe you are going to transform.

Returns Schema like object, should not be None or empty

Return type Any

on_init(*df*)

Callback for initializing physical partition that contains one or multiple logical partitions. You may put expensive initialization logic here so you will not have to repeat that in *transform()*

Note:

- This call can be on a random machine (depending on the ExecutionEngine you use), you should get the context from the properties of this class
- You can get physical partition no (if available from the execution egnine) from *cursor()*
- The input dataframe may be unbounded, but must be empty aware. That means you must not consume the df by any means, and you can not count. However you can safely peek the first row of the dataframe for multiple times.
- The input dataframe is never empty. Empty physical partitions are skipped

---

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the entire dataframe of this physical partition

Return type None

transform(*df*)

The transformation logic from one local dataframe to another local dataframe.

Note:

- This function operates on logical partition level
- This call can be on a random machine (depending on the ExecutionEngine you use), you should get the *context* from the properties of this class
- The input dataframe may be unbounded, but must be empty aware. It's safe to consume it for ONLY ONCE
- The input dataframe is never empty. Empty dataframes are skipped

---

Parameters **df** (fugue.dataframe.dataframe.LocalDataFrame) – one logical partition to transform on

Returns transformed dataframe

Return type *fugue.dataframe.dataframe.LocalDataFrame*

fugue.extensions.context

class fugue.extensions.context.**ExtensionContext**

Bases: object

Context variables that extensions can access. It's also the base class of all extensions.

property callback: *fugue.rpc.base.RPCClient*

RPC client to talk to driver, this is for transformers only, and available on both driver and workers

property cursor: *fugue.collections.partition.PartitionCursor*

Cursor of the current logical partition, this is for transformers only, and only available on worker side

property execution_engine: *fugue.execution.execution_engine.ExecutionEngine*

Execution engine for the current execution, this is only available on driver side

property has_callback: bool

Whether this transformer has callback

property key_schema: triad.collections.schema.Schema

Partition keys schema, this is for transformers only, and available on both driver and workers

property output_schema: triad.collections.schema.Schema

Output schema of the operation. This is accessible for all extensions (if defined), and on both driver and workers

property params: triad.collections.dict.ParamDict

Parameters set for using this extension.

Examples

```
>>> FugueWorkflow().df(...).transform(using=dummy, params={"a": 1})
```

You will get `{"a": 1}` as *params* in the `dummy` transformer

property partition_spec: *fugue.collections.partition.PartitionSpec*
Partition specification, this is for all extensions except for creators, and available on both driver and workers

validate_on_compile()

Return type None

validate_on_runtime(*data*)

Parameters **data** (*Union[fugue.dataframe.dataframe.DataFrame, fugue.dataframe.dataframes.DataFrames]*) –

Return type None

property validation_rules: **Dict[str, Any]**
Extension input validation rules defined by user

property workflow_conf: *triad.collections.dict.ParamDict*
Workflow level configs, this is accessible even in *Transformer* and *CoTransformer*

Examples

```
>>> dag = FugueWorkflow().df(...).transform(using=dummy)
>>> dag.run(NativeExecutionEngine(conf={"b": 10}))
```

You will get `{"b": 10}` as *workflow_conf* in the `dummy` transformer on both driver and workers.

4.1.6 fugue.rpc

fugue.rpc.base

class fugue.rpc.base.**EmptyRPCHandler**
Bases: *fugue.rpc.base.RPCHandler*

The class representing empty *RPCHandler*

class fugue.rpc.base.**NativeRPCClient**(*server*, *key*)
Bases: *fugue.rpc.base.RPCClient*

Native RPC Client that can only be used locally. Use *make_client()* to create this instance.

Parameters

- **server** (NativeRPCServer) – the parent *NativeRPCServer*
- **key** (*str*) – the unique key for the handler and this client

class fugue.rpc.base.**NativeRPCServer**(*conf*)
Bases: *fugue.rpc.base.RPCServer*

Native RPC Server that can only be used locally.

Parameters **conf** (*Any*) – the Fugue Configuration Tutorial

make_client(*handler*)
Add handler and correspondent *NativeRPCClient*

Parameters **handler** (*Any*) – RPChandler like object

Returns the native RPC client

Return type *fugue.rpc.base.RPCClient*

start_server()
Do nothing

Return type None

stop_server()
Do nothing

Return type None

class fugue.rpc.base.**RPCClient**
Bases: object

RPC client interface

class fugue.rpc.base.**RPCFunc**(*func*)
Bases: *fugue.rpc.base.RPCHandler*

RPCHandler wrapping a python function.

Parameters **func** (*Callable*) – a python function

class fugue.rpc.base.**RPCHandler**
Bases: *fugue.rpc.base.RPCClient*

RPC handler hosting the real logic on driver side

property running: bool
Whether the handler is in running state

start()
Start the handler, wrapping *start_handler()*

Returns the instance itself

Return type *fugue.rpc.base.RPCHandler*

start_handler()
User implementation of starting the handler

Return type None

stop()
Stop the handler, wrapping *stop_handler()*

Return type None

stop_handler()
User implementation of stopping the handler

Return type None

class fugue.rpc.base.**RPCServer**(*conf*)
Bases: *fugue.rpc.base.RPCHandler*, abc.ABC

Server abstract class hosting multiple *RPCHandler*.

Parameters **conf** (*Any*) – the Fugue Configuration Tutorial

property conf: triad.collections.dict.ParamDict
Config initialized this instance

invoke(*key*, **args*, ***kwargs*)
Invoke the correspondent handler

Parameters

- **key** (*str*) – key of the handler
- **args** (*Any*) –
- **kwargs** (*Any*) –

Returns the return value of the handler

Return type Any

abstract make_client(*handler*)
Make a *RPCHandler* and return the correspondent *RPCClient*

Parameters **handler** (*Any*) – RPChandler like object

Returns the client connecting to the handler

Return type *fugue.rpc.base.RPCClient*

register(*handler*)
Register the hander into the server

Parameters **handler** (*Any*) – RPChandler like object

Returns the unique key of the handler

Return type str

start_handler()
Wrapper to start the server, do not override or call directly

Return type None

abstract start_server()
User implementation of starting the server

Return type None

stop_handler()
Wrapper to stop the server, do not override or call directly

Return type None

abstract stop_server()
User implementation of stopping the server

Return type None

fugue.rpc.base.**make_rpc_server**(*conf*)
Make *RPCServer* based on configuration. If '*fugue.rpc.server*` is set, then the value will be used as the server type for the initialization. Otherwise, a *NativeRPCServer* instance will be returned

Parameters **conf** (*Any*) – the Fugue Configuration Tutorial

Returns the RPC server

Return type *fugue.rpc.base.RPCServer*

fugue.rpc.base.**to_rpc_handler**(*obj*)
Convert object to *RPCHandler*. If the object is already RPCHandler, then the original instance will be returned. If the object is None then *EmptyRPCHandler* will be returned. If the object is a python function then *RPCFunc* will be returned.

Parameters **obj** (*Any*) – RPChandler like object

Returns the RPC handler

Return type *fugue.rpc.base.RPCHandler*

fugue.rpc.flask

class fugue.rpc.flask.**FlaskRPCClient**(*key*, *host*, *port*, *timeout_sec*)
Bases: *fugue.rpc.base.RPCClient*

Flask RPC Client that can be used distributedly. Use *make_client()* to create this instance.

Parameters

- **key** (*str*) – the unique key for the handler and this client
- **host** (*str*) – the host address of the flask server
- **port** (*int*) – the port of the flask server
- **timeout_sec** (*float*) – timeout seconds for flask clients

class fugue.rpc.flask.**FlaskRPCServer**(*conf*)
Bases: *fugue.rpc.base.RPCServer*

Flask RPC server that can be used in a distributed environment. It's required to set fugue.rpc.flask_server.host and fugue.rpc.flask_server.port. If fugue.rpc.flask_server.timeout is not set, then the client could hang until the connection is closed.

Parameters **conf** (*Any*) – the Fugue Configuration Tutorial

---

Examples

```
conf = {
    "fugue.rpc.server": "fugue.rpc.flask.FlaskRPCServer",
    "fugue.rpc.flask_server.host": "127.0.0.1",
    "fugue.rpc.flask_server.port": "1234",
    "fugue.rpc.flask_server.timeout": "2 sec",
}

with make_rpc_server(conf).start() as server:
    server...
```

---

make_client(*handler*)
Add handler and correspondent *FlaskRPCClient*

Parameters **handler** (*Any*) – RPChandler like object

Returns the flask RPC client that can be distributed

Return type *fugue.rpc.base.RPCClient*

start_server()
Start Flask RPC server

Return type None

`stop_server()`
Stop Flask RPC server

Return type None

4.1.7 fugue.workflow

fugue.workflow.module

`fugue.workflow.module.module(func=None, as_method=False, name=None, on_dup='overwrite')`
Decorator for module

Please read Module Tutorial

Parameters

- **func** (*Optional[Callable]*) –
- **as_method** (*bool*) –
- **name** (*Optional[str]*) –
- **on_dup** (*str*) –

Return type Any

fugue.workflow.utils

`fugue.workflow.utils.is_acceptable_raw_df(df)`
Whether the input `df` can be converted to *WorkflowDataFrame*

Parameters **df** (*Any*) – input raw dataframe

Returns whether this dataframe is convertible

Return type bool

`fugue.workflow.utils.register_raw_df_type(df_type)`
Register a base type of dataframe that can be recognized by *FugueWorkflow* and converted to *WorkflowDataFrame*

Parameters **df_type** (*Type*) – dataframe type, for example `dask.dataframe.DataFrame`

Return type None

fugue.workflow.workflow

`class fugue.workflow.workflow.FugueWorkflow(*args, **kwargs)`
Bases: `object`

Fugue Workflow, also known as the Fugue Programming Interface.

In Fugue, we use DAG to represent workflows, DAG construction and execution are different steps, this class is mainly used in the construction step, so all things you added to the workflow is **description** and they are not executed until you call *run()*

Read this to learn how to initialize it in different ways and pros and cons.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

add(*task*, **args*, ***kwargs*)

This method should not be called directly by users. Use *create()*, *process()*, *output()* instead

Parameters

- **task** (*fugue.workflow._tasks.FugueTask*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type *fugue.workflow.workflow.WorkflowDataFrame*

assert_eq(**dfs*, ***params*)

Compare if these dataframes are equal. It's for internal, unit test purpose only. It will convert both dataframes to *LocalBoundedDataFrame*, so it assumes all dataframes are small and fast enough to convert. DO NOT use it on critical or expensive tasks.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **digits** – precision on float number comparison, defaults to 8
- **check_order** – if to compare the row orders, defaults to False
- **check_schema** – if compare schemas, defaults to True
- **check_content** – if to compare the row values, defaults to True
- **check_metadata** – if to compare the dataframe metadatas, defaults to True
- **no_pandas** – if true, it will compare the string representations of the dataframes, otherwise, it will convert both to pandas dataframe to compare, defaults to False
- **params** (*Any*) –

Raises **AssertionError** – if not equal

Return type None

assert_not_eq(**dfs*, ***params*)

Assert if all dataframes are not equal to the first dataframe. It's for internal, unit test purpose only. It will convert both dataframes to *LocalBoundedDataFrame*, so it assumes all dataframes are small and fast enough to convert. DO NOT use it on critical or expensive tasks.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **digits** – precision on float number comparison, defaults to 8
- **check_order** – if to compare the row orders, defaults to False
- **check_schema** – if compare schemas, defaults to True
- **check_content** – if to compare the row values, defaults to True
- **check_metadata** – if to compare the dataframe metadatas, defaults to True
- **no_pandas** – if true, it will compare the string representations of the dataframes, otherwise, it will convert both to pandas dataframe to compare, defaults to False
- **params** (*Any*) –

Raises **AssertionError** – if any dataframe equals to the first dataframe

Return type None

property conf: triad.collections.dict.ParamDict
All configs of this workflow and underlying *ExecutionEngine* (if given)

create(*using*, *schema=None*, *params=None*)
Run a creator to create a dataframe.

Please read the Creator Tutorial

Parameters

- **using** (*Any*) – creator-like object, if it is a string, then it must be the alias of a registered creator
- **schema** (*Optional[Any]*) – Schema like object, defaults to None. The creator will be able to access this value from *output_schema()*
- **params** (*Optional[Any]*) – Parameters like object to run the creator, defaults to None. The creator will be able to access this value from *params()*
- **pre_partition** – Partition like object, defaults to None. The creator will be able to access this value from *partition_spec()*

Returns result dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

create_data(*data*, *schema=None*, *metadata=None*, *data_determiner=None*)
Create dataframe.

Parameters

- **data** (*Any*) – DataFrame like object or `Yielded`
- **schema** (*Optional[Any]*) – Schema like object, defaults to None
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None
- **data_determiner** (*Optional[Callable[[Any], str]]*) – a function to compute unique id from `data`

Returns a dataframe of the current workflow

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note: By default, the input `data` does not affect the determinism of the workflow (but `schema` and `etadata` do), because the amount of compute can be unpredictable. But if you want `data` to affect the determinism of the workflow, you can provide the function to compute the unique id of `data` using `data_determiner`

---

df(*data*, *schema=None*, *metadata=None*, *data_determiner=None*)
Create dataframe. Alias of *create_data()*

Parameters

- **data** (*Any*) – DataFrame like object or `Yielded`
- **schema** (*Optional[Any]*) – Schema like object, defaults to None
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None

- **data_determiner** (*Optional[Callable[[Any], str]]*) – a function to compute unique id from `data`

Returns a dataframe of the current workflow

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note: By default, the input `data` does not affect the determinism of the workflow (but `schema` and `etadata` do), because the amount of compute can be unpredictable. But if you want `data` to affect the determinism of the workflow, you can provide the function to compute the unique id of `data` using `data_determiner`

`get_result`(*df*)

After *run()*, get the result of a dataframe defined in the dag

Returns a calculated dataframe

Parameters **df** (`fugue.workflow.workflow.WorkflowDataFrame`) –

Return type *fugue.dataframe.dataframe.DataFrame*

Examples

```
dag = FugueWorkflow()
df1 = dag.df([[0]],"a:int")
dag.run()
dag.get_result(df1).show()
```

`intersect`(**dfs*, *distinct=True*)

Intersect dataframes in `dfs`.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **distinct** (*bool*) – whether to perform *distinct* after intersection, default to True

Returns intersected dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note: Currently, all dataframes in `dfs` must have identical schema, otherwise exception will be thrown.

`join`(**dfs*, *how*, *on=None*)

Join dataframes. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **how** (*str*) – can accept `semi`, `left_semi`, `anti`, `left_anti`, `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None

Returns joined dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

load(*path*, *fmt=''*, *columns=None*, ***kwargs*)
Load dataframe from persistent storage. Read this for details.

Parameters

- **path** (*str*) – file path
- **fmt** (*str*) – format hint can accept `parquet`, `csv`, `json`, defaults to "", meaning to infer
- **columns** (*Optional[Any]*) – list of columns or a Schema like object, defaults to None
- **kwargs** (*Any*) –

Returns dataframe from the file

Return type *WorkflowDataFrame*

out_transform(**dfs*, *using*, *params=None*, *pre_partition=None*, *ignore_errors=[]*, *callback=None*)
Transform dataframes using transformer, it materializes the execution immediately and returns nothing

Please read the Transformer Tutorial

Parameters

- **dfs** (*Any*) – DataFrames like object
- **using** (*Any*) – transformer-like object, if it is a string, then it must be the alias of a registered output transformer/cotransformer
- **schema** – Schema like object, defaults to None. The transformer will be able to access this value from *output_schema()*
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The transformer will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. It's recommended to use the equivalent wayt, which is to call *partition()* and then call *out_transform()* without this parameter
- **ignore_errors** (*List[Any]*) – list of exception types the transformer can ignore, defaults to empty list
- **callback** (*Optional[Any]*) – RPChandler like object, defaults to None

Return type None

Note: *transform()* can be lazy and will return the transformed dataframe, *out_transform()* is guaranteed to execute immediately (eager) and return nothing

---

output(**dfs*, *using*, *params=None*, *pre_partition=None*)
Run a outputter on dataframes.

Please read the Outputter Tutorial

Parameters

- **using** (*Any*) – outputter-like object, if it is a string, then it must be the alias of a registered outputter
- **params** (*Optional[Any]*) – Parameters like object to run the outputter, defaults to None. The outputter will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. The outputter will be able to access this value from *partition_spec()*

- **dfs** (*Any*) –

Return type None

process(**dfs*, *using*, *schema=None*, *params=None*, *pre_partition=None*)

Run a processor on the dataframes.

Please read the Processor Tutorial

Parameters

- **dfs** (*Any*) – DataFrames like object
- **using** (*Any*) – processor-like object, if it is a string, then it must be the alias of a registered processor
- **schema** (*Optional[Any]*) – Schema like object, defaults to None. The processor will be able to access this value from *output_schema()*
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The processor will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. The processor will be able to access this value from *partition_spec()*

Returns result dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

run(**args*, ***kwargs*)

Execute the workflow and compute all dataframes. If not arguments, it will use *NativeExecutionEngine* to run the workflow.

Examples

```
dag = FugueWorkflow()
df1 = dag.df([[0]],"a:int").transform(a_transformer)
df2 = dag.df([[0]],"b:int")

dag.run(SparkExecutionEngine)
df1.result.show()
df2.result.show()

dag = FugueWorkflow()
df1 = dag.df([[0]],"a:int").transform(a_transformer)
df1.yield_dataframe_as("x")

result = dag.run(SparkExecutionEngine)
result["x"]  # SparkDataFrame
```

Read this to learn how to run in different ways and pros and cons.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type *fugue.dataframe.dataframes.DataFrames*

select(**statements*, *sql_engine=None*, *sql_engine_params=None*)
Execute SELECT statement using *SQLEngine*

Parameters

- **statements** (*Any*) – a list of sub-statements in string or *WorkflowDataFrame*
- **sql_engine** (*Optional[Any]*) – it can be empty string or null (use the default SQL engine), a string (use the registered SQL engine), an *SQLEngine* type, or the *SQLEngine* instance (you can use `None` to use the default one), defaults to None
- **sql_engine_params** (*Optional[Any]*) –

Returns result of the SELECT statement

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Examples

```
with FugueWorkflow() as dag:
    a = dag.df([[0,"a"]],a:int,b:str)
    b = dag.df([[0]],a:int)
    c = dag.select("SELECT a FROM",a,"UNION SELECT * FROM",b)
```

Please read this for more examples

set_op(*how*, **dfs*, *distinct=True*)
Union, subtract or intersect dataframes.

Parameters

- **how** (*str*) – can accept `union`, `left_semi`, `anti`, `left_anti`, `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`
- **dfs** (*Any*) – DataFrames like object
- **distinct** (*bool*) – whether to perform *distinct* after the set operation, default to True

Returns result dataframe of the set operation

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note: Currently, all dataframes in `dfs` must have identical schema, otherwise exception will be thrown.

---

show(**dfs*, *rows=10*, *show_count=False*, *title=None*)
Show the dataframes. See examples.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **rows** (*int*) – max number of rows, defaults to 10
- **show_count** (*bool*) – whether to show total count, defaults to False
- **title** (*Optional[str]*) – title to display on top of the dataframe, defaults to None
- **best_width** – max width for the output table, defaults to 100

Return type None

Note:

- When you call this method, it means you want the dataframe to be printed when the workflow executes. So the dataframe won't show until you run the workflow.
- When `show_count` is True, it can trigger expensive calculation for a distributed dataframe. So if you call this function directly, you may need to *persist()* the dataframe. Or you can turn on Auto Persist

spec_uuid()
UUID of the workflow spec (*description*)

Return type str

subtract(**dfs*, *distinct=True*)
Subtract `dfs[1:]` from `dfs[0]`.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **distinct** (*bool*) – whether to perform *distinct* after subtraction, default to True

Returns subtracted dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note: Currently, all dataframes in `dfs` must have identical schema, otherwise exception will be thrown.

transform(**dfs*, *using*, *schema=None*, *params=None*, *pre_partition=None*, *ignore_errors=[]*, *callback=None*)
Transform dataframes using transformer.

Please read the Transformer Tutorial

Parameters

- **dfs** (*Any*) – DataFrames like object
- **using** (*Any*) – transformer-like object, if it is a string, then it must be the alias of a registered transformer/cotransformer
- **schema** (*Optional[Any]*) – Schema like object, defaults to None. The transformer will be able to access this value from *output_schema()*
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The transformer will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. It's recommended to use the equivalent wayt, which is to call *partition()* and then call *transform()* without this parameter
- **ignore_errors** (*List[Any]*) – list of exception types the transformer can ignore, defaults to empty list
- **callback** (*Optional[Any]*) – RPChandler like object, defaults to None

Returns the transformed dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note: *transform()* can be lazy and will return the transformed dataframe, *out_transform()* is guaranteed to execute immediately (eager) and return nothing

union(**dfs*, *distinct=True*)
Union dataframes in `dfs`.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **distinct** (*bool*) – whether to perform *distinct* after union, default to True

Returns unioned dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note: Currently, all dataframes in `dfs` must have identical schema, otherwise exception will be thrown.

property yields: Dict[str, *fugue.collections.yielded.Yielded*]

zip(**dfs*, *how='inner'*, *partition=None*, *temp_path=None*, *to_file_threshold=-1*)
Zip multiple dataframes together with given partition specifications.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **how** (*str*) – can accept `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`, defaults to `inner`
- **partition** (*Optional[Any]*) – Partition like object, defaults to None.
- **temp_path** (*Optional[str]*) – file path to store the data (used only if the serialized data is larger than `to_file_threshold`), defaults to None
- **to_file_threshold** (*Any*) – file byte size threshold, defaults to -1

Returns a zipped dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Note:

- If `dfs` is dict like, the zipped dataframe will be dict like, If `dfs` is list like, the zipped dataframe will be list like
- It's fine to contain only one dataframe in `dfs`

See also:

Read CoTransformer and Zip & Comap for details

class fugue.workflow.workflow.**WorkflowDataFrame**(*workflow*, *task*, *metadata=None*)
Bases: *fugue.dataframe.dataframe.DataFrame*

It represents the edges in the graph constructed by *FugueWorkflow*. In Fugue, we use DAG to represent workflows, and the edges are strictly dataframes. DAG construction and execution are different steps, this class is used in the construction step. Although it inherits from *DataFrame*, it's not concerete data. So a lot of the operations are not allowed. If you want to obtain the concrete Fugue *DataFrame*, use *compute()* to execute the workflow.

Normally, you don't construct it by yourself, you will just use the methods of it.

Parameters

- **workflow** (FugueWorkflow) – the parent workflow it belongs to
- **task** (*fugue.workflow._tasks.FugueTask*) – the task that generates this dataframe
- **metadata** (*Any*) – dict-like metadata, defaults to None

aggregate(**agg_cols*, ***kwagg_cols*)

Aggregate on dataframe

Parameters

- **df** – the dataframe to aggregate on
- **agg_cols** (fugue.column.expressions.ColumnExpr) – aggregation expressions
- **kwagg_cols** (fugue.column.expressions.ColumnExpr) – aggregation expressions to be renamed to the argument names
- **self** (*fugue.workflow.workflow.TDF*) –

Returns the aggregated result as a dataframe

Return type fugue.workflow.workflow.TDF

New Since

0.6.0

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
import fugue.column.functions as f

# SELECT MAX(b) AS b FROM df
df.aggregate(f.max(col("b")))

# SELECT a, MAX(b) AS x FROM df GROUP BY a
df.partition_by("a").aggregate(f.max(col("b")).alias("x"))
df.partition_by("a").aggregate(x=f.max(col("b")))
```

---

alter_columns(*columns*)

Change column types

Parameters

- **columns** (*Any*) – Schema like object
- **self** (*fugue.workflow.workflow.TDF*) –

Returns a new dataframe with the new column types

Return type *WorkflowDataFrame*

Note: The output dataframe will not change the order of original schema.

Examples

```
>>> df.alter_columns("a:int,b;str")
```

anti_join(**dfs*, *on=None*)

LEFT ANTI Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

as_array(*columns=None*, *type_safe=False*)

Raises **NotImplementedError** – don't call this method

Parameters

- **columns** (*Optional[List[str]]*) –
- **type_safe** (*bool*) –

Return type List[Any]

as_array_iterable(*columns=None*, *type_safe=False*)

Raises **NotImplementedError** – don't call this method

Parameters

- **columns** (*Optional[List[str]]*) –
- **type_safe** (*bool*) –

Return type Iterable[Any]

as_local()

Raises **NotImplementedError** – don't call this method

Return type *fugue.dataframe.dataframe.DataFrame*

assert_eq(**dfs*, ***params*)

Wrapper of *fugue.workflow.workflow.FugueWorkflow.assert_eq()* to compare this dataframe with other dataframes.

Parameters

- **dfs** (*Any*) – DataFrames like object

- **digits** – precision on float number comparison, defaults to 8
- **check_order** – if to compare the row orders, defaults to False
- **check_schema** – if compare schemas, defaults to True
- **check_content** – if to compare the row values, defaults to True
- **check_metadata** – if to compare the dataframe metadatas, defaults to True
- **no_pandas** – if true, it will compare the string representations of the dataframes, otherwise, it will convert both to pandas dataframe to compare, defaults to False
- **params** (*Any*) –

Raises **AssertionError** – if not equal

Return type None

assert_not_eq(**dfs*, ***params*)

Wrapper of *fugue.workflow.workflow.FugueWorkflow.assert_not_eq()* to compare this dataframe with other dataframes.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **digits** – precision on float number comparison, defaults to 8
- **check_order** – if to compare the row orders, defaults to False
- **check_schema** – if compare schemas, defaults to True
- **check_content** – if to compare the row values, defaults to True
- **check_metadata** – if to compare the dataframe metadatas, defaults to True
- **no_pandas** – if true, it will compare the string representations of the dataframes, otherwise, it will convert both to pandas dataframe to compare, defaults to False
- **params** (*Any*) –

Raises **AssertionError** – if any dataframe is equal to the first dataframe

Return type None

assign(**args*, ***kwargs*)

Update existing columns with new values and add new columns

Parameters

- **df** – the dataframe to set columns
- **args** (fugue.column.expressions.ColumnExpr) – column expressions
- **kwargs** (*Any*) – column expressions to be renamed to the argument names, if a value is not *ColumnExpr*, it will be treated as a literal
- **self** (*fugue.workflow.workflow.TDF*) –

Returns a new dataframe with the updated values

Return type fugue.workflow.workflow.TDF

Tip: This can be used to cast data types, alter column values or add new columns. But you can't use aggregation in columns.

New Since

0.6.0

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
from fugue import FugueWorkflow

dag = FugueWorkflow()
df = dag.df(pandas_df)

# add/set 1 as column x
df.assign(lit(1,"x"))
df.assign(x=1)

# add/set x to be a+b
df.assign((col("a")+col("b")).alias("x"))
df.assign(x=col("a")+col("b"))

# cast column a data type to double
df.assign(col("a").cast(float))

# cast + new columns
df.assign(col("a").cast(float),x=1,y=col("a")+col("b"))
```

broadcast()
Broadcast the current dataframe

Returns the broadcasted dataframe

Return type *WorkflowDataFrame*

Parameters **self** (*fugue.workflow.workflow.TDF*) –

checkpoint()

Parameters **self** (*fugue.workflow.workflow.TDF*) –

Return type fugue.workflow.workflow.TDF

compute(**args*, ***kwargs*)
Trigger the parent workflow to *run()* and to generate and cache the result dataframe this instance represent.

Examples

```
>>> df = FugueWorkflow().df([[0]],"a:int").transform(a_transformer)
>>> df.compute().as_pandas()  # pandas dataframe
>>> df.compute(SparkExecutionEngine).native  # spark dataframe
```

Note: Consider using *fugue.workflow.workflow.FugueWorkflow.run()* instead. Because this method actually triggers the entire workflow to run, so it may be confusing to use this method because extra time may be taken to compute unrelated dataframes.

```
dag = FugueWorkflow()
df1 = dag.df([[0]],"a:int").transform(a_transformer)
df2 = dag.df([[0]],"b:int")

dag.run(SparkExecutionEngine)
df1.result.show()
df2.result.show()
```

Return type *fugue.dataframe.dataframe.DataFrame*

count()

Raises **NotImplementedError** – don't call this method

Return type int

cross_join(**dfs*)

CROSS Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

deterministic_checkpoint(*lazy=False*, *partition=None*, *single=False*, *namespace=None*, ***kwargs*)

Cache the dataframe as a temporary file

Parameters

- **lazy** (*bool*) – whether it is a lazy checkpoint, defaults to False (eager)
- **partition** (*Optional[Any]*) – Partition like object, defaults to None.
- **single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – paramteters for the underlying execution engine function
- **namespace** (*Optional[Any]*) – a value to control determinism, defaults to None.
- **self** (*fugue.workflow.workflow.TDF*) –

Returns the cached dataframe

Return type fugue.workflow.workflow.TDF

Note: The difference vs *strong_checkpoint()* is that this checkpoint is not removed after execution, so it can take effect cross execution if the dependent compute logic is not changed.

distinct()
Get distinct dataframe. Equivalent to `SELECT DISTINCT * FROM df`

Returns dataframe with unique records

Parameters **self** (*fugue.workflow.workflow.TDF*) –

Return type fugue.workflow.workflow.TDF

drop(*columns*, *if_exists=False*)
Drop columns from the dataframe.

Parameters

- **columns** (*List[str]*) – columns to drop
- **if_exists** (*bool*) – if setting to True, it will ignore non-existent columns, defaults to False
- **self** (*fugue.workflow.workflow.TDF*) –

Returns the dataframe after dropping columns

Return type *WorkflowDataFrame*

dropna(*how='any'*, *thresh=None*, *subset=None*)
Drops records containing NA records

Parameters

- **how** (*str*) – 'any' or 'all'. 'any' drops rows that contain any nulls. 'all' drops rows that contain all nulls.
- **thresh** (*Optional[int]*) – int, drops rows that have less than thresh non-null values
- **subset** (*Optional[List[str]]*) – list of columns to operate on
- **self** (*fugue.workflow.workflow.TDF*) –

Returns dataframe with incomplete records dropped

Return type fugue.workflow.workflow.TDF

property empty: bool

Raises **NotImplementedError** – don't call this method

fillna(*value*, *subset=None*)
Fills NA values with replacement values

Parameters

- **value** (*Any*) – if scalar, fills all columns with same value. if dictionary, fills NA using the keys as column names and the values as the replacement values.
- **subset** (*Optional[List[str]]*) – list of columns to operate on. ignored if value is a dictionary
- **self** (*fugue.workflow.workflow.TDF*) –

Returns dataframe with NA records filled

Return type fugue.workflow.workflow.TDF

filter(*condition*)
Filter rows by the given condition

Parameters

- **df** – the dataframe to be filtered
- **condition** (fugue.column.expressions.ColumnExpr) – (boolean) column expression
- **self** (*fugue.workflow.workflow.TDF*) –

Returns a new filtered dataframe

Return type fugue.workflow.workflow.TDF

New Since

0.6.0

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
import fugue.column.functions as f
from fugue import FugueWorkflow

dag = FugueWorkflow()
df = dag.df(pandas_df)

df.filter((col("a")>1) & (col("b")=="x"))
df.filter(f.coalesce(col("a"),col("b"))>1)
```

---

full_outer_join(**dfs*, *on=None*)

CROSS Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

inner_join(**dfs*, *on=None*)

INNER Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

intersect(**dfs*, *distinct=True*)

Intersect this dataframe with `dfs`.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **distinct** (*bool*) – whether to perform *distinct* after intersection, default to True
- **self** (*fugue.workflow.workflow.TDF*) –

Returns intersected dataframe

Return type fugue.workflow.workflow.TDF

Note: Currently, all dataframes in `dfs` must have identical schema, otherwise exception will be thrown.

---

property is_bounded: bool

Raises **NotImplementedError** – don't call this method

property is_local: bool

Raises **NotImplementedError** – don't call this method

join(**dfs*, *how*, *on=None*)

Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **how** (*str*) – can accept `semi`, `left_semi`, `anti`, `left_anti`, `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

left_anti_join(**dfs*, *on=None*)

LEFT ANTI Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

left_outer_join(**dfs*, *on=None*)
LEFT OUTER Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

left_semi_join(**dfs*, *on=None*)
LEFT SEMI Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

property name: str
Name of its task spec

property num_partitions: int

Raises **NotImplementedError** – don't call this method

out_transform(*using*, *params=None*, *pre_partition=None*, *ignore_errors=[]*, *callback=None*)
Transform this dataframe using transformer. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.out_transform()*

Please read the Transformer Tutorial

Parameters

- **using** (*Any*) – transformer-like object, if it is a string, then it must be the alias of a registered output transformer/cotransformer
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The transformer will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. It's recommended to use the equivalent wayt, which is to call *partition()* and then call *transform()* without this parameter
- **ignore_errors** (*List[Any]*) – list of exception types the transformer can ignore, defaults to empty list
- **callback** (*Optional[Any]*) – RPChandler like object, defaults to None
- **self** (*fugue.workflow.workflow.TDF*) –

Return type None

Note: *transform()* can be lazy and will return the transformed dataframe, *out_transform()* is guaranteed to execute immediately (eager) and return nothing

output(*using*, *params=None*, *pre_partition=None*)
Run a outputter on this dataframe. It's a simple wrapper of *fugue.workflow.workflow.FugueWorkflow.output()*

Please read the Outputter Tutorial

Parameters

- **using** (*Any*) – outputter-like object, if it is a string, then it must be the alias of a registered outputter
- **params** (*Optional[Any]*) – Parameters like object to run the outputter, defaults to None. The outputter will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. The outputter will be able to access this value from *partition_spec()*

Return type None

partition(**args*, ***kwargs*)
Partition the current dataframe. Please read the Partition Tutorial

Parameters

- **args** (*Any*) – Partition like object
- **kwargs** (*Any*) – Partition like object
- **self** (*fugue.workflow.workflow.TDF*) –

Returns dataframe with the partition hint

Return type *WorkflowDataFrame*

Note: Normally this step is fast because it's to add a partition hint for the next step.

partition_by(**keys*, ***kwargs*)
Partition the current dataframe by keys. Please read the Partition Tutorial. This is a wrapper of *partition()*

Parameters

- **keys** (*str*) – partition keys
- **kwargs** (*Any*) – Partition like object excluding `by` and `partition_by`
- **self** (*fugue.workflow.workflow.TDF*) –

Returns dataframe with the partition hint

Return type *WorkflowDataFrame*

property partition_spec: *fugue.collections.partition.PartitionSpec*
The partition spec set on the dataframe for next steps to use

Examples

```
dag = FugueWorkflow()
df = dag.df([[0],[1]], "a:int")
assert df.partition_spec.empty
df2 = df.partition(by=["a"])
assert df.partition_spec.empty
assert df2.partition_spec == PartitionSpec(by=["a"])
```

peek_array()

Raises **NotImplementedError** – don't call this method

Return type Any

per_partition_by(**keys*)

Partition the current dataframe by keys so each physical partition contains only one logical partition. Please read the Partition Tutorial. This is a wrapper of *partition()*

Parameters

- **keys** (*str*) – partition keys
- **self** (*fugue.workflow.workflow.TDF*) –

Returns dataframe that is both logically and physically partitioned by `keys`

Return type *WorkflowDataFrame*

Note: This is a hint but not enforced, certain execution engines will not respect this hint.

per_row()

Partition the current dataframe to one row per partition. Please read the Partition Tutorial. This is a wrapper of *partition()*

Returns dataframe that is evenly partitioned by row count

Return type *WorkflowDataFrame*

Parameters **self** (*fugue.workflow.workflow.TDF*) –

Note: This is a hint but not enforced, certain execution engines will not respect this hint.

persist()

Persist the current dataframe

Returns the persisted dataframe

Return type *WorkflowDataFrame*

Parameters **self** (*fugue.workflow.workflow.TDF*) –

Note: `persist` can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

persist method is considered as weak checkpoint. Sometimes, it may be necessary to use strong checkpoint, which is *checkpoint()*

---

process(*using*, *schema=None*, *params=None*, *pre_partition=None*)

Run a processor on this dataframe. It's a simple wrapper of *fugue.workflow.workflow.FugueWorkflow.process()*

Please read the Processor Tutorial

Parameters

- **using** (*Any*) – processor-like object, if it is a string, then it must be the alias of a registered processor
- **schema** (*Optional[Any]*) – Schema like object, defaults to None. The processor will be able to access this value from *output_schema()*
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The processor will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. The processor will be able to access this value from *partition_spec()*
- **self** (*fugue.workflow.workflow.TDF*) –

Returns result dataframe

Return type *WorkflowDataFrame*

rename(**args*, ***kwargs*)

Rename the dataframe using a mapping dict

Parameters

- **args** (*Any*) – list of dicts containing rename maps
- **kwargs** (*Any*) – rename map
- **self** (*fugue.workflow.workflow.TDF*) –

Returns a new dataframe with the new names

Return type *WorkflowDataFrame*

Note: This interface is more flexible than *fugue.dataframe.dataframe.DataFrame.rename()*

Examples

```
>>> df.rename({"a": "b"}, c="d", e="f")
```

---

property result: *fugue.dataframe.dataframe.DataFrame*

The concrete DataFrame obtained from *compute()*. This property will not trigger compute again, but compute should have been called earlier and the result is cached.

right_outer_join(**dfs*, *on=None*)

RIGHT OUTER Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

sample(*n=None*, *frac=None*, *replace=False*, *seed=None*)
Sample dataframe by number of rows or by fraction

Parameters

- **n** (*Optional[int]*) – number of rows to sample, one and only one of `n` and `fact` must be set
- **frac** (*Optional[float]*) – fraction [0,1] to sample, one and only one of `n` and `fact` must be set
- **replace** (*bool*) – whether replacement is allowed. With replacement, there may be duplicated rows in the result, defaults to False
- **seed** (*Optional[int]*) – seed for randomness, defaults to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns sampled dataframe

Return type fugue.workflow.workflow.TDF

save(*path*, *fmt=''*, *mode='overwrite'*, *partition=None*, *single=False*, ***kwargs*)
Save this dataframe to a persistent storage

Parameters

- **path** (*str*) – output path
- **fmt** (*str*) – format hint can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition** (*Optional[Any]*) – Partition like object, how to partition the dataframe before saving, defaults to empty
- **single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Save & Load.

save_and_use(*path*, *fmt=''*, *mode='overwrite'*, *partition=None*, *single=False*, ***kwargs*)
Save this dataframe to a persistent storage and load back to use in the following steps

Parameters

- **path** (*str*) – output path
- **fmt** (*str*) – format hint can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"

- **partition** (*Optional[Any]*) – Partition like object, how to partition the dataframe before saving, defaults to empty
- **single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework
- **self** (*fugue.workflow.workflow.TDF*) –

Return type fugue.workflow.workflow.TDF

For more details and examples, read Save & Load.

property schema: triad.collections.schema.Schema

Raises **NotImplementedError** – don't call this method

select(**columns*, *where=None*, *having=None*, *distinct=False*)

The functional interface for SQL select statement

Parameters

- **columns** (*Union[str, fugue.column.expressions.ColumnExpr]*) – column expressions, for strings they will represent the column names
- **where** (*Optional[fugue.column.expressions.ColumnExpr]*) – `WHERE` condition expression, defaults to None
- **having** (*Optional[fugue.column.expressions.ColumnExpr]*) – `having` condition expression, defaults to None. It is used when `cols` contains aggregation columns, defaults to None
- **distinct** (*bool*) – whether to return distinct result, defaults to False
- **self** (*fugue.workflow.workflow.TDF*) –

Returns the select result as a new dataframe

Return type fugue.workflow.workflow.TDF

New Since

0.6.0

Attention: This interface is experimental, it's subjected to change in new versions.

See also:

Please find more expression examples in *fugue.column.sql* and *fugue.column.functions*

Examples

```
import fugue.column.functions as f
from fugue import FugueWorkflow

dag = FugueWorkflow()
df = dag.df(pandas_df)

# select existed and new columns
```

(continues on next page)

(continued from previous page)

```
df.select("a","b",lit(1,"another")))
df.select("a",(col("b")+lit(1)).alias("x"))

# select distinct
df.select("a","b",lit(1,"another")),distinct=True)

# aggregation
# SELECT COUNT(DISTINCT *) AS x FROM df
df.select(f.count_distinct(col("*")).alias("x"))

# SELECT a, MAX(b+1) AS x FROM df GROUP BY a
df.select("a",f.max(col("b")+lit(1)).alias("x"))

# SELECT a, MAX(b+1) AS x FROM df
#   WHERE b<2 AND a>1
#   GROUP BY a
#   HAVING MAX(b+1)>0
df.select(
    "a",f.max(col("b")+lit(1)).alias("x"),
    where=(col("b")<2) & (col("a")>1),
    having=f.max(col("b")+lit(1))>0
)
```

---

semi_join(**dfs*, *on=None*)

LEFT SEMI Join this dataframe with dataframes. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.join()*. Read Join tutorials on workflow and engine for details

Parameters

- **dfs** (*Any*) – DataFrames like object
- **on** (*Optional[Iterable[str]]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys. Default to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns joined dataframe

Return type *WorkflowDataFrame*

show(*rows=10*, *show_count=False*, *title=None*, *best_width=100*)

Show the dataframe. See examples.

Parameters

- **rows** (*int*) – max number of rows, defaults to 10
- **show_count** (*bool*) – whether to show total count, defaults to False
- **title** (*Optional[str]*) – title to display on top of the dataframe, defaults to None
- **best_width** (*int*) – max width for the output table, defaults to 100

Return type None

Note:

- When you call this method, it means you want the dataframe to be printed when the workflow executes. So the dataframe won't show until you run the workflow.
- When `show_count` is True, it can trigger expensive calculation for a distributed dataframe. So if you call this function directly, you may need to *persist()* the dataframe. Or you can turn on tutorial:tutorials/advanced/useful_config:auto persist

---

spec_uuid()
UUID of its task spec

Return type str

strong_checkpoint(*lazy=False*, *partition=None*, *single=False*, ***kwargs*)
Cache the dataframe as a temporary file

Parameters

- **lazy** (*bool*) – whether it is a lazy checkpoint, defaults to False (eager)
- **partition** (*Optional[Any]*) – Partition like object, defaults to None.
- **single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – paramteters for the underlying execution engine function
- **self** (*fugue.workflow.workflow.TDF*) –

Returns the cached dataframe

Return type fugue.workflow.workflow.TDF

Note: Strong checkpoint guarantees the output dataframe compute dependency is from the temporary file. Use strong checkpoint only when *weak_checkpoint()* can't be used.

Strong checkpoint file will be removed after the execution of the workflow.

---

subtract(**dfs*, *distinct=True*)
Subtract `dfs` from this dataframe.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **distinct** (*bool*) – whether to perform *distinct* after subtraction, default to True
- **self** (*fugue.workflow.workflow.TDF*) –

Returns subtracted dataframe

Return type fugue.workflow.workflow.TDF

Note: Currently, all dataframes in `dfs` must have identical schema, otherwise exception will be thrown.

---

take(*n*, *presort=None*, *na_position='last'*)
Get the first n rows of a DataFrame per partition. If a presort is defined, use the presort before applying take. presort overrides partition_spec.presort

Parameters

- **n** (*int*) – number of rows to return
- **presort** (*Optional[str]*) – presort expression similar to partition presort

- **na_position** (*str*) – position of null values during the presort. can accept `first` or `last`
- **self** (*fugue.workflow.workflow.TDF*) –

Returns n rows of DataFrame per partition

Return type fugue.workflow.workflow.TDF

transform(*using*, *schema=None*, *params=None*, *pre_partition=None*, *ignore_errors=[]*, *callback=None*)

Transform this dataframe using transformer. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.transform()*

Please read the Transformer Tutorial

Parameters

- **using** (*Any*) – transformer-like object, if it is a string, then it must be the alias of a registered transformer/cotransformer
- **schema** (*Optional[Any]*) – Schema like object, defaults to None. The transformer will be able to access this value from *output_schema()*
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The transformer will be able to access this value from *params()*
- **pre_partition** (*Optional[Any]*) – Partition like object, defaults to None. It's recommended to use the equivalent wayt, which is to call *partition()* and then call *transform()* without this parameter
- **ignore_errors** (*List[Any]*) – list of exception types the transformer can ignore, defaults to empty list
- **callback** (*Optional[Any]*) – RPChandler like object, defaults to None
- **self** (*fugue.workflow.workflow.TDF*) –

Returns the transformed dataframe

Return type *WorkflowDataFrame*

Note: *transform()* can be lazy and will return the transformed dataframe, *out_transform()* is guaranteed to execute immediately (eager) and return nothing

---

union(**dfs*, *distinct=True*)

Union this dataframe with `dfs`.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **distinct** (*bool*) – whether to perform *distinct* after union, default to True
- **self** (*fugue.workflow.workflow.TDF*) –

Returns unioned dataframe

Return type fugue.workflow.workflow.TDF

Note: Currently, all dataframes in `dfs` must have identical schema, otherwise exception will be thrown.

---

weak_checkpoint(*lazy=False*, ***kwargs*)
Cache the dataframe in memory

Parameters

- **lazy** (*bool*) – whether it is a lazy checkpoint, defaults to False (eager)
- **kwargs** (*Any*) – paramteters for the underlying execution engine function
- **self** (*fugue.workflow.workflow.TDF*) –

Returns the cached dataframe

Return type fugue.workflow.workflow.TDF

Note: Weak checkpoint in most cases is the best choice for caching a dataframe to avoid duplicated computation. However it does not guarantee to break up the the compute dependency for this dataframe, so when you have very complicated compute, you may encounter issues such as stack overflow. Also, weak checkpoint normally caches the dataframe in memory, if memory is a concern, then you should consider *strong_checkpoint()*

---

property workflow: *fugue.workflow.workflow.FugueWorkflow*
The parent workflow

yield_dataframe_as(*name*, *as_local=False*)
Yield a dataframe that can be accessed without the current execution engine

Parameters

- **name** (*str*) – the name of the yielded dataframe
- **as_local** (*bool*) – yield the local version of the dataframe
- **self** (*fugue.workflow.workflow.TDF*) –

Return type None

Note: When `as_local` is True, it can trigger an additional compute to do the conversion. To avoid recompute, you should add `persist` before yielding.

---

yield_file_as(*name*)
Cache the dataframe in file

Parameters

- **name** (*str*) – the name of the yielded dataframe
- **self** (*fugue.workflow.workflow.TDF*) –

Return type None

Note: In only the following cases you can yield file:

- you have not checkpointed (persisted) the dataframe, for example `df.yield_file_as("a")`
- you have used *deterministic_checkpoint()*, for example `df.deterministic_checkpoint().yield_file_as("a")`
- yield is workflow, compile level logic

For the first case, the yield will also be a strong checkpoint so whenever you yield a dataframe as a file, the dataframe has been saved as a file and loaded back as a new dataframe.

---

zip(**dfs*, *how='inner'*, *partition=None*, *temp_path=None*, *to_file_threshold=-1*)

Zip this data frame with multiple dataframes together with given partition specifications. It's a wrapper of *fugue.workflow.workflow.FugueWorkflow.zip()*.

Parameters

- **dfs** (*Any*) – DataFrames like object
- **how** (*str*) – can accept `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`, defaults to `inner`
- **partition** (*Optional[Any]*) – Partition like object, defaults to None.
- **temp_path** (*Optional[str]*) – file path to store the data (used only if the serialized data is larger than `to_file_threshold`), defaults to None
- **to_file_threshold** (*Any*) – file byte size threshold, defaults to -1
- **self** (*fugue.workflow.workflow.TDF*) –

Returns a zipped dataframe

Return type *WorkflowDataFrame*

Note:

- `dfs` must be list like, the zipped dataframe will be list like
- `dfs` is fine to be empty
- If you want dict-like zip, use *fugue.workflow.workflow.FugueWorkflow.zip()*

See also:

Read CoTransformer and Zip & Comap for details

class fugue.workflow.workflow.**WorkflowDataFrames**(**args*, ***kwargs*)

Bases: *fugue.dataframe.dataframes.DataFrames*

Ordered dictionary of WorkflowDataFrames. There are two modes: with keys and without keys. If without key _<n> will be used as the key for each dataframe, and it will be treated as an array in Fugue framework.

It's immutable, once initialized, you can't add or remove element from it.

It's a subclass of *DataFrames*, but different from DataFrames, in the initialization you should always use *WorkflowDataFrame*, and they should all come from the same *FugueWorkflow*.

Examples

```
dag = FugueWorkflow()
df1 = dag.df([[0]],"a:int").transform(a_transformer)
df2 = dag.df([[0]],"b:int")
dfs1 = WorkflowDataFrames(df1, df2)  # as array
dfs2 = WorkflowDataFrames([df1, df2])  # as array
dfs3 = WorkflowDataFrames(a=df1, b=df2)  # as dict
dfs4 = WorkflowDataFrames(dict(a=df1, b=df2))  # as dict
dfs5 = WorkflowDataFrames(dfs4, c=df2)  # copy and update
```

(continues on next page)

(continued from previous page)

```
dfs5["b"].show()  # how you get element when it's a dict
dfs1[0].show()  # how you get element when it's an array
```

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

property workflow: *fugue.workflow.workflow.FugueWorkflow*
The parent workflow

4.1.8 fugue.constants

fugue.constants.**register_global_conf**(*conf*, *on_dup=0*)
Register global Fugue configs that can be picked up by any Fugue execution engines as the base configs.

Parameters

- **conf** (*Dict[str, Any]*) – the config dictionary
- **on_dup** (*int*) – see `triad.collections.dict.ParamDict.update()`, defaults to `ParamDict.OVERWRITE`

Return type None

Note: When using `ParamDict.THROW` or `on_dup`, it's transactional. If any key in `conf` is already in global config and the value is different from the new value, then ValueError will be thrown.

Examples

```
from fugue import register_global_conf, NativeExecutionEngine

register_global_conf({"my.value",1})

engine = NativeExecutionEngine()
assert 1 == engine.conf["my.value"]

engine = NativeExecutionEngine({"my.value",2})
assert 2 == engine.conf["my.value"]
```

4.1.9 fugue.exceptions

exception fugue.exceptions.**FugueBug**
Bases: *fugue.exceptions.FugueError*

Fugue internal bug

exception fugue.exceptions.**FugueDataFrameEmptyError**
Bases: *fugue.exceptions.FugueDataFrameError*

Fugue dataframe is empty

exception fugue.exceptions.**FugueDataFrameError**
Bases: *fugue.exceptions.FugueError*

Fugue dataframe related error

exception fugue.exceptions.**FugueDataFrameInitError**
Bases: *fugue.exceptions.FugueDataFrameError*

Fugue dataframe initialization error

exception fugue.exceptions.**FugueDataFrameOperationError**
Bases: *fugue.exceptions.FugueDataFrameError*

Fugue dataframe invalid operation

exception fugue.exceptions.**FugueError**
Bases: Exception

Fugue exceptions

exception fugue.exceptions.**FugueInterfacelessError**
Bases: *fugue.exceptions.FugueWorkflowCompileError*

Fugue interfaceless exceptions

exception fugue.exceptions.**FuguePluginsRegistrationError**
Bases: *fugue.exceptions.FugueError*

Fugue plugins registration error

exception fugue.exceptions.**FugueWorkflowCompileError**
Bases: *fugue.exceptions.FugueWorkflowError*

Fugue workflow compile time error

exception fugue.exceptions.**FugueWorkflowCompileValidationError**
Bases: *fugue.exceptions.FugueWorkflowCompileError*

Fugue workflow compile time validation error

exception fugue.exceptions.**FugueWorkflowError**
Bases: *fugue.exceptions.FugueError*

Fugue workflow exceptions

exception fugue.exceptions.**FugueWorkflowRuntimeError**
Bases: *fugue.exceptions.FugueWorkflowError*

Fugue workflow compile time error

exception fugue.exceptions.**FugueWorkflowRuntimeValidationError**
Bases: *fugue.exceptions.FugueWorkflowRuntimeError*

Fugue workflow runtime validation error

4.1.10 fugue.interfaceless

fugue.interfaceless.**out_transform**(*df*, *using*, *params=None*, *partition=None*, *callback=None*, *ignore_errors=None*, *engine=None*, *engine_conf=None*)

Transform this dataframe using transformer. It's a wrapper of *out_transform()* and *run()*. It let you do the basic dataframe transformation without using *FugueWorkflow* and *DataFrame*. The input can be native type only

Please read the Transformer Tutorial

Parameters

- **df** (*Any*) – DataFrame like object or `Yielded` or a path string to a parquet file
- **using** (*Any*) – transformer-like object, can't be a string expression
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The transformer will be able to access this value from *params()*
- **partition** (*Optional[Any]*) – Partition like object, defaults to None.
- **callback** (*Optional[Any]*) – RPChandler like object, defaults to None
- **ignore_errors** (*Optional[List[Any]]*) – list of exception types the transformer can ignore, defaults to None (empty list)
- **engine** (*Optional[Any]*) – it can be empty string or null (use the default execution engine), a string (use the registered execution engine), an *ExecutionEngine* type, or the *ExecutionEngine* instance , or a tuple of two values where the first value represents execution engine and the second value represents the sql engine (you can use `None` for either of them to use the default one), defaults to None
- **engine_conf** (*Optional[Any]*) – Parameters like object, defaults to None

Return type None

Note: This function can only take parquet file paths in *df*. Csv and other file formats are disallowed.

This transformation is guaranteed to execute immediately (eager) and return nothing

---

fugue.interfaceless.**transform**(*df*, *using*, *schema=None*, *params=None*, *partition=None*, *callback=None*, *ignore_errors=None*, *engine=None*, *engine_conf=None*, *force_output_fugue_dataframe=False*, *persist=False*, *as_local=False*, *save_path=None*, *checkpoint=False*)

Transform this dataframe using transformer. It's a wrapper of *transform()* and *run()*. It let you do the basic dataframe transformation without using *FugueWorkflow* and *DataFrame*. Both input and output can be native types only.

Please read the Transformer Tutorial

Parameters

- **df** (*Any*) – DataFrame like object or `Yielded` or a path string to a parquet file
- **using** (*Any*) – transformer-like object, can't be a string expression
- **schema** (*Optional[Any]*) – Schema like object, defaults to None. The transformer will be able to access this value from *output_schema()*
- **params** (*Optional[Any]*) – Parameters like object to run the processor, defaults to None. The transformer will be able to access this value from *params()*

- **partition** (*Optional[Any]*) – Partition like object, defaults to None
- **callback** (*Optional[Any]*) – RPChandler like object, defaults to None
- **ignore_errors** (*Optional[List[Any]]*) – list of exception types the transformer can ignore, defaults to None (empty list)
- **engine** (*Optional[Any]*) – it can be empty string or null (use the default execution engine), a string (use the registered execution engine), an *ExecutionEngine* type, or the *ExecutionEngine* instance , or a tuple of two values where the first value represents execution engine and the second value represents the sql engine (you can use `None` for either of them to use the default one), defaults to None
- **engine_conf** (*Optional[Any]*) – Parameters like object, defaults to None
- **force_output_fugue_dataframe** (*bool*) – If true, the function will always return a `FugueDataFrame`, otherwise, if `df` is in native dataframe types such as pandas dataframe, then the output will also in its native format. Defaults to False
- **persist** (*bool*) – Whether to persist(materialize) the dataframe before returning
- **as_local** (*bool*) – If true, the result will be converted to a `LocalDataFrame`
- **save_path** (*Optional[str]*) – Whether to save the output to a file (see the note)
- **checkpoint** (*bool*) – Whether to add a checkpoint for the output (see the note)

Returns the transformed dataframe, if `df` is a native dataframe (e.g. pd.DataFrame, spark dataframe, etc), the output will be a native dataframe, the type is determined by the execution engine you use. But if `df` is of type *DataFrame*, then the output will also be a *DataFrame*

Return type Any

Note: This function may be lazy and return the transformed dataframe.

Note: When you use callback in this function, you must be careful that the output dataframe must be materialized. Otherwise, if the real compute happens out of the function call, the callback receiver is already shut down. To do that you can either use `persist` or `as_local`, both will materialize the dataframe before the callback receiver shuts down.

Note:

- When *save_path* is None and *checkpoint* is False, then the output will not be saved into a file. The return will be a dataframe.
- When *save_path* is None and *checkpoint* is True, then the output will be saved into the path set by *fugue.workflow.checkpoint.path*, the name will be randomly chosen, and it is NOT a deterministic checkpoint, so if you run multiple times, the output will be saved into different files. The return will be a dataframe.
- When *save_path* is not None and *checkpoint* is False, then the output will be saved into *save_path*. The return will be the value of *save_path*
- When *save_path* is not None and *checkpoint* is True, then the output will be saved into *save_path*. The return will be the dataframe from *save_path*

This function can only take parquet file paths in *df* and *save_path*. Csv and other file formats are disallowed.

The checkpoint here is NOT deterministic, so re-run will generate new checkpoints.

If you want to read and write other file formats or if you want to use deterministic checkpoints, please use *FugueWorkflow*.

4.1.11 fugue.registry

4.2 fugue_sql

4.2.1 fugue_sql.exceptions

exception fugue_sql.exceptions.**FugueSQLError**
Bases: *fugue.exceptions.FugueWorkflowCompileError*

Fugue SQL error

exception fugue_sql.exceptions.**FugueSQLRuntimeError**
Bases: *fugue.exceptions.FugueWorkflowRuntimeError*

Fugue SQL runtime error

exception fugue_sql.exceptions.**FugueSQLSyntaxError**
Bases: *fugue_sql.exceptions.FugueSQLError*

Fugue SQL syntax error

4.2.2 fugue_sql.workflow

class fugue_sql.workflow.**FugueSQLWorkflow**(**args*, ***kwargs*)
Bases: *fugue.workflow.workflow.FugueWorkflow*

Fugue workflow that supports Fugue SQL. Please read the Fugue SQL Tutorial.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

property sql_vars: Dict[str, *fugue.workflow.workflow.WorkflowDataFrame*]

fugue_sql.workflow.**fsql**(*sql*, **args*, *fsql_ignore_case=False*, ***kwargs*)
Fugue SQL functional interface

Parameters

- **sql** (*str*) – the Fugue SQL string (can be a jinja template)
- **args** (*Any*) – variables related to the SQL string
- **fsql_ignore_case** (*bool*) – whether to ignore case when parsing the SQL string defaults to False.
- **kwargs** (*Any*) – variables related to the SQL string

Returns the translated Fugue workflow

Return type *fugue_sql.workflow.FugueSQLWorkflow*

```
# Basic case
fsql('''
CREATE [[0]] SCHEMA a:int
PRINT
''').run()

# With external data sources
df = pd.DataFrame([[0],[1]], columns=["a"])
fsql('''
SELECT * FROM df WHERE a=0
PRINT
''').run()

# With external variables
df = pd.DataFrame([[0],[1]], columns=["a"])
t = 1
fsql('''
SELECT * FROM df WHERE a={{t}}
PRINT
''').run()

# The following is the explicit way to specify variables and datafrems
# (recommended)
df = pd.DataFrame([[0],[1]], columns=["a"])
t = 1
fsql('''
SELECT * FROM df WHERE a={{t}}
PRINT
''', df=df, t=t).run()

# Using extensions
def dummy(df:pd.DataFrame) -> pd.DataFrame:
    return df

fsql('''
CREATE [[0]] SCHEMA a:int
TRANSFORM USING dummy SCHEMA *
PRINT
''').run()

# It's recommended to provide full path of the extension inside
# Fugue SQL, so the SQL definition and exeuction can be more
# independent from the extension definition.

# Run with different execution engines
sql = '''
CREATE [[0]] SCHEMA a:int
TRANSFORM USING dummy SCHEMA *
PRINT
'''

fsql(sql).run(user_defined_spark_session())
fsql(sql).run(SparkExecutionEngine, {"spark.executor.instances":10})
```

(continues on next page)

(continued from previous page)

```
fsql(sql).run(DaskExecutionEngine)

# Passing dataframes between fsql calls
result = fsql('''
CREATE [[0]] SCHEMA a:int
YIELD DATAFRAME AS x

CREATE [[1]] SCHEMA a:int
YIELD DATAFRAME AS y
''').run(DaskExecutionEngine)

fsql('''
SELECT * FROM x
UNION
SELECT * FROM y
UNION
SELECT * FROM z

PRINT
''', result, z=pd.DataFrame([[2]], columns=["z"])).run()

# Get framework native dataframes
result["x"].native  # Dask dataframe
result["y"].native  # Dask dataframe
result["x"].as_pandas()  # Pandas dataframe

# Use lower case fugue sql
df = pd.DataFrame([[0],[1]], columns=["a"])
t = 1
fsql('''
select * from df where a={{t}}
print
''', df=df, t=t, fsql_ignore_case=True).run()
```

4.3 fugue_duckdb

4.3.1 fugue_duckdb.dask

class fugue_duckdb.dask.**DuckDaskExecutionEngine**(*conf=None*, *connection=None*, *dask_client=None*)

Bases: *fugue_duckdb.execution_engine.DuckExecutionEngine*

A hybrid engine of DuckDB and Dask. Most operations will be done by DuckDB, but for `map`, it will use Dask to fully utilize local CPUs. The engine can be used with a real Dask cluster, but practically, this is more useful for local process.

Parameters

- **conf** (*Any*) – Parameters like object, read the Fugue Configuration Tutorial to learn Fugue specific options
- **connection** (*Optional[duckdb.DuckDBPyConnection]*) – DuckDB connection
- **dask_client** (*Optional[distributed.client.Client]*) –

broadcast(*df*)

Broadcast the dataframe to all workers for a distributed computing framework

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe

Returns the broadcasted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

convert_yield_dataframe(*df*, *as_local*)

Convert a yield dataframe to a dataframe that can be used after this execution engine stops.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **as_local** (*bool*) – whether yield a local dataframe

Returns another DataFrame that can be used after this execution engine stops

Return type *fugue.dataframe.dataframe.DataFrame*

Note: By default, the output dataframe is the input dataframe. But it should be overridden if when an engine stops and the input dataframe will become invalid.

For example, if you custom a spark engine where you start and stop the spark session in this engine's *start_engine()* and *stop_engine()*, then the spark dataframe will be invalid. So you may consider converting it to a local dataframe so it can still exist after the engine stops.

---

property dask_client: distributed.client.Client

map(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)

Apply a function to each partition after you partition the data in a specified way.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframe.LocalDataFrame*]*, fugue.dataframe.dataframe.LocalDataFrame*]*) – the function to apply on every logical partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframe.DataFrame*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the map operation

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing, you must read this to understand what map is used for and how it should work.

---

persist(*df*, *lazy=False*, ***kwargs*)

Force materializing and caching the dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe
- **lazy** (*bool*) – `True`: first usage of the output will trigger persisting to happen; `False` (eager): persist is forced to happend immediately. Default to `False`
- **args** – parameter to pass to the underlying persist implementation
- **kwargs** (*Any*) – parameter to pass to the underlying persist implementation

Returns the persisted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: `persist` can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

---

repartition(*df*, *partition_spec*)

Partition the input dataframe using `partition_spec`.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how you want to partition the dataframe

Returns repartitioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing please read the Partition Tutorial

---

save_df(*df*, *path*, *format_hint=None*, *mode='overwrite'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *force_single=False*, ***kwargs*)

Save dataframe to a persistent storage

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **path** (*str*) – output path
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how to partition the dataframe before saving, defaults to empty
- **force_single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Load & Save.

to_df(*df*, *schema=None*, *metadata=None*)
Convert a data structure to this engine compatible DataFrame

Parameters

- **data** – *DataFrame*, pandas DataFramme or list or iterable of arrays or others that is supported by certain engine implementation
- **schema** (*Optional[Any]*) – Schema like object, defaults to None
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None
- **df** (*Any*) –

Returns engine compatible dataframe

Return type *fugue_duckdb.dataframe.DuckDataFrame*

Note: There are certain conventions to follow for a new implementation:

- if the input is already in compatible dataframe type, it should return itself
- all other methods in the engine interface should take arbitrary dataframes and call this method to convert before doing anything

---

4.3.2 fugue_duckdb.dataframe

class fugue_duckdb.dataframe.**DuckDataFrame**(*rel*, *metadata=None*)
Bases: *fugue.dataframe.dataframe.LocalBoundedDataFrame*

DataFrame that wraps DuckDB `DuckDBPyRelation`.

Parameters

- **rel** (*duckdb.DuckDBPyRelation*) – `DuckDBPyRelation` object
- **metadata** (*Any*) – dict-like object with string keys, default `None`

alter_columns(*columns*)
Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)
Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

`as_array_iterable`(*columns=None*, *type_safe=False*)
Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

`as_arrow`(*type_safe=False*)
Convert to pyArrow DataFrame

Parameters **type_safe** (*bool*) –

Return type pyarrow.lib.Table

`as_local`()
Always return self, because it's a LocalDataFrame

Return type *fugue.dataframe.dataframe.LocalDataFrame*

`as_pandas`()
Convert to pandas DataFrame

Return type pandas.core.frame.DataFrame

`count`()
Get number of rows of this dataframe

Return type int

`property empty: bool`
Whether this dataframe is empty

`head`(*n*, *columns=None*)
Get first n rows of the dataframe as 2-dimensional array

Parameters

- **n** (*int*) – number of rows
- **columns** (*Optional[List[str]]*) – selected columns, defaults to None (all columns)

Returns 2-dimensional array

Return type List[Any]

`property native: duckdb.DuckDBPyRelation`
DuckDB relation object

`peek_array`()
Peek the first row of the dataframe as array

Raises ***FugueDataFrameEmptyError*** – if it is empty

Return type Any

rename(*columns*)

Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

4.3.3 fugue_duckdb.execution_engine

class fugue_duckdb.execution_engine.**DuckDBEngine**(*execution_engine*)

Bases: *fugue.execution.execution_engine.SQLEngine*

DuckDB SQL backend implementation.

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) – the execution engine this sql engine will run on

Return type None

select(*dfs*, *statement*)

Execute select statement on the sql engine.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **statement** (*str*) – the SELECT statement using the dfs keys as tables

Returns result of the SELECT statement

Return type *fugue.dataframe.dataframe.DataFrame*

Examples

```
>>> dfs = DataFrames(a=df1, b=df2)
>>> sql_engine.select(dfs, "SELECT * FROM a UNION SELECT * FROM b")
```

Note: There can be tables that is not in dfs. For example you want to select from hive without input DataFrames:

```
>>> sql_engine.select(DataFrames(), "SELECT * FROM hive.a.table")
```

---

class fugue_duckdb.execution_engine.**DuckExecutionEngine**(*conf=None*, *connection=None*)

Bases: *fugue.execution.execution_engine.ExecutionEngine*

The execution engine using DuckDB. Please read the ExecutionEngine Tutorial to understand this important Fugue concept

Parameters

- **conf** (*Any*) – Parameters like object, read the Fugue Configuration Tutorial to learn Fugue specific options

- **connection** (*Optional[duckdb.DuckDBPyConnection]*) – DuckDB connection

broadcast(*df*)

Broadcast the dataframe to all workers for a distributed computing framework

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe

Returns the broadcasted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

property connection: duckdb.DuckDBPyConnection

convert_yield_dataframe(*df*, *as_local*)

Convert a yield dataframe to a dataframe that can be used after this execution engine stops.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **as_local** (*bool*) – whether yield a local dataframe

Returns another DataFrame that can be used after this execution engine stops

Return type *fugue.dataframe.dataframe.DataFrame*

Note: By default, the output dataframe is the input dataframe. But it should be overridden if when an engine stops and the input dataframe will become invalid.

For example, if you custom a spark engine where you start and stop the spark session in this engine's *start_engine()* and *stop_engine()*, then the spark dataframe will be invalid. So you may consider converting it to a local dataframe so it can still exist after the engine stops.

---

property default_sql_engine: *fugue.execution.execution_engine.SQLEngine*

Default SQLEngine if user doesn't specify

distinct(*df*, *metadata=None*)

Equivalent to `SELECT DISTINCT * FROM df`

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – dataframe
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns [description]

Return type *DataFrame*

dropna(*df*, *how='any'*, *thresh=None*, *subset=None*, *metadata=None*)

Drop NA recods from dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **how** (*str*) – 'any' or 'all'. 'any' drops rows that contain any nulls. 'all' drops rows that contain all nulls.
- **thresh** (*Optional[int]*) – int, drops rows that have less than thresh non-null values
- **subset** (*Optional[List[str]]*) – list of columns to operate on

- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records dropped

Return type *DataFrame*

fillna(*df*, *value*, *subset=None*, *metadata=None*)
Fill NULL, NAN, NAT values in a dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **value** (*Any*) – if scalar, fills all columns with same value. if dictionary, fills NA using the keys as column names and the values as the replacement values.
- **subset** (*Optional[List[str]]*) – list of columns to operate on. ignored if value is a dictionary
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records filled

Return type *DataFrame*

property fs: triad.collections.fs.FileSystem
File system of this engine instance

intersect(*df1*, *df2*, *distinct=True*, *metadata=None*)
Intersect df1 and df2

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – true for INTERSECT (== INTERSECT DISTINCT), false for INTERSECT ALL
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of df1 and df2 must be identical, or an exception will be thrown.

---

join(*df1*, *df2*, *how*, *on=[]*, *metadata=None*)
Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **how** (*str*) – can accept semi, left_semi, anti, left_anti, inner, left_outer, right_outer, full_outer, cross
- **on** (*List[str]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys.

- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the joined dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Please read this

---

load_df(*path*, *format_hint=None*, *columns=None*, ***kwargs*)
Load dataframe from persistent storage

Parameters

- **path** (*Union[str, List[str]]*) – the path to the dataframe
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **columns** (*Optional[Any]*) – list of columns or a Schema like object, defaults to None
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Returns an engine compatible dataframe

Return type *fugue.dataframe.dataframe.LocalBoundedDataFrame*

For more details and examples, read Zip & Comap.

property log: logging.Logger
Logger of this engine instance

map(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)
Apply a function to each partition after you partition the data in a specified way.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframe.LocalDataFrame*]*, fugue.dataframe.dataframe.LocalDataFrame*]*) – the function to apply on every logical partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframe.DataFrame*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the map operation

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing, you must read this to understand what map is used for and how it should work.

`persist`(*df*, *lazy=False*, ***kwargs*)
Force materializing and caching the dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe
- **lazy** (*bool*) – `True`: first usage of the output will trigger persisting to happen; `False` (eager): persist is forced to happend immediately. Default to `False`
- **args** – parameter to pass to the underlying persist implementation
- **kwargs** (*Any*) – parameter to pass to the underlying persist implementation

Returns the persisted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: `persist` can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

`repartition`(*df*, *partition_spec*)
Partition the input dataframe using `partition_spec`.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how you want to partition the dataframe

Returns repartitioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing please read the Partition Tutorial

`sample`(*df*, *n=None*, *frac=None*, *replace=False*, *seed=None*, *metadata=None*)
Sample dataframe by number of rows or by fraction

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*Optional[int]*) – number of rows to sample, one and only one of `n` and `frac` must be set
- **frac** (*Optional[float]*) – fraction [0,1] to sample, one and only one of `n` and `frac` must be set
- **replace** (*bool*) – whether replacement is allowed. With replacement, there may be duplicated rows in the result, defaults to False
- **seed** (*Optional[int]*) – seed for randomness, defaults to None
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns sampled dataframe

Return type *DataFrame*

save_df(*df*, *path*, *format_hint=None*, *mode='overwrite'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *force_single=False*, ***kwargs*)

Save dataframe to a persistent storage

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **path** (*str*) – output path
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how to partition the dataframe before saving, defaults to empty
- **force_single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Load & Save.

stop()

Stop this execution engine, do not override You should customize *stop_engine()* if necessary.

Return type None

subtract(*df1*, *df2*, *distinct=True*, *metadata=None*)

`df1 - df2`

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `EXCEPT` (== `EXCEPT DISTINCT`), `false` for `EXCEPT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

take(*df*, *n*, *presort*, *na_position='last'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *metadata=None*)

Get the first n rows of a DataFrame per partition. If a presort is defined, use the presort before applying take. presort overrides partition_spec.presort. The Fugue implementation of the presort follows Pandas convention of specifying NULLs first or NULLs last. This is different from the Spark and SQL convention of NULLs as the smallest value.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*int*) – number of rows to return

- **presort** (*str*) – presort expression similar to partition presort
- **na_position** (*str*) – position of null values during the presort. can accept `first` or `last`
- **partition_spec** (fugue.collections.partition.PartitionSpec) – PartitionSpec to apply the take operation
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns n rows of DataFrame per partition

Return type *DataFrame*

to_df(*df*, *schema=None*, *metadata=None*)

Convert a data structure to this engine compatible DataFrame

Parameters

- **data** – *DataFrame*, pandas DataFramme or list or iterable of arrays or others that is supported by certain engine implementation
- **schema** (*Optional[Any]*) – Schema like object, defaults to None
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None
- **df** (*Any*) –

Returns engine compatible dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: There are certain conventions to follow for a new implementation:

- if the input is already in compatible dataframe type, it should return itself
- all other methods in the engine interface should take arbitrary dataframes and call this method to convert before doing anything

---

union(*df1*, *df2*, *distinct=True*, *metadata=None*)

Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `UNION` (== `UNION DISTINCT`), `false` for `UNION ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

4.3.4 fugue_duckdb.ibis_engine

class fugue_duckdb.ibis_engine.**DuckDBIbisEngine**(*execution_engine*)
Bases: *fugue_ibis.execution.ibis_engine.IbisEngine*

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) –

Return type None

select(*dfs*, *ibis_func*)
Execute the ibis select expression.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **ibis_func** (*Callable[[ibis.backends.base.BaseBackend], ibis.expr.types.TableExpr]*) – the ibis compute function

Returns result of the ibis function

Return type *fugue.dataframe.dataframe.DataFrame*

Note: This interface is experimental, so it is subjected to change.

4.3.5 fugue_duckdb.registry

4.4 fugue_spark

4.4.1 fugue_spark.dataframe

class fugue_spark.dataframe.**SparkDataFrame**(*df=None*, *schema=None*, *metadata=None*)
Bases: *fugue.dataframe.dataframe.DataFrame*

DataFrame that wraps Spark DataFrame. Please also read the DataFrame Tutorial to understand this Fugue concept

Parameters

- **df** (*Any*) – pyspark.sql.DataFrame
- **schema** (*Any*) – Schema like object or pyspark.sql.types.StructType, defaults to None.
- **metadata** (*Any*) – Parameters like object, defaults to None

Note:

- You should use *fugue_spark.execution_engine.SparkExecutionEngine.to_df()* instead of construction it by yourself.
- If schema is set, then there will be type cast on the Spark DataFrame if the schema is different.

alter_columns(*columns*)
Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)
Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

---

as_array_iterable(*columns=None*, *type_safe=False*)
Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If `type_safe` is False, then the returned values are 'raw' values.

---

as_local()
Convert this dataframe to a *LocalDataFrame*

Return type *fugue.dataframe.dataframe.LocalDataFrame*

as_pandas()
Convert to pandas DataFrame

Return type pandas.core.frame.DataFrame

count()
Get number of rows of this dataframe

Return type int

property empty: bool
Whether this dataframe is empty

head(*n*, *columns=None*)
Get first n rows of the dataframe as 2-dimensional array

Parameters

- **n** (*int*) – number of rows
- **columns** (*Optional[List[str]]*) – selected columns, defaults to None (all columns)

Returns 2-dimensional array

Return type List[Any]

property is_bounded: bool
Whether this dataframe is bounded

property is_local: bool
Whether this dataframe is a *LocalDataFrame*

property native: pyspark.sql.dataframe.DataFrame
The wrapped Spark DataFrame

Return type pyspark.sql.DataFrame

property num_partitions: int
Number of physical partitions of this dataframe. Please read the Partition Tutorial

peek_array()
Peek the first row of the dataframe as array

Raises *FugueDataFrameEmptyError* – if it is empty

Return type List[Any]

rename(*columns*)
Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

4.4.2 fugue_spark.execution_engine

class fugue_spark.execution_engine.**SparkExecutionEngine**(*spark_session=None*, *conf=None*)
Bases: *fugue.execution.execution_engine.ExecutionEngine*

The execution engine based on SparkSession.

Please read the ExecutionEngine Tutorial to understand this important Fugue concept

Parameters

- **spark_session** (*Optional[pyspark.sql.session.SparkSession]*) – Spark session, defaults to None to get the Spark session by `getOrCreate()`
- **conf** (*Any*) – Parameters like object defaults to None, read the Fugue Configuration Tutorial to learn Fugue specific options

broadcast(*df*)
Broadcast the dataframe to all workers for a distributed computing framework

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe

Returns the broadcasted dataframe

Return type *fugue_spark.dataframe.SparkDataFrame*

property default_sql_engine: *fugue.execution.execution_engine.SQLEngine*
Default SQLEngine if user doesn't specify

distinct(*df*, *metadata=None*)
Equivalent to `SELECT DISTINCT * FROM df`

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – dataframe
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns [description]

Return type *DataFrame*

dropna(*df*, *how='any'*, *thresh=None*, *subset=None*, *metadata=None*)
Drop NA recods from dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **how** (*str*) – 'any' or 'all'. 'any' drops rows that contain any nulls. 'all' drops rows that contain all nulls.
- **thresh** (*Optional[int]*) – int, drops rows that have less than thresh non-null values
- **subset** (*Optional[List[str]]*) – list of columns to operate on
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records dropped

Return type *DataFrame*

fillna(*df*, *value*, *subset=None*, *metadata=None*)
Fill `NULL`, `NAN`, `NAT` values in a dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **value** (*Any*) – if scalar, fills all columns with same value. if dictionary, fills NA using the keys as column names and the values as the replacement values.
- **subset** (*Optional[List[str]]*) – list of columns to operate on. ignored if value is a dictionary
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records filled

Return type *DataFrame*

property fs: triad.collections.fs.FileSystem
File system of this engine instance

intersect(*df1*, *df2*, *distinct=True*, *metadata=None*)
Intersect `df1` and `df2`

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe

- **distinct** (*bool*) – `true` for `INTERSECT (== INTERSECT DISTINCT)`, `false` for `INTERSECT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

join(*df1*, *df2*, *how*, *on=[]*, *metadata=None*)
Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **how** (*str*) – can accept `semi`, `left_semi`, `anti`, `left_anti`, `inner`, `left_outer`, `right_outer`, `full_outer`, `cross`
- **on** (*List[str]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys.
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the joined dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Please read *this*

---

load_df(*path*, *format_hint=None*, *columns=None*, ***kwargs*)
Load dataframe from persistent storage

Parameters

- **path** (*Union[str, List[str]]*) – the path to the dataframe
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **columns** (*Optional[Any]*) – list of columns or a Schema like object, defaults to None
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Returns an engine compatible dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

For more details and examples, read Zip & Comap.

property log: logging.Logger
Logger of this engine instance

map(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)
Apply a function to each partition after you partition the data in a specified way.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframe.LocalDataFrame*], *fugue.dataframe.dataframe.LocalDataFrame*]*) – the function to apply on every logical partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframe.DataFrame*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the map operation

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing, you must read this to understand what map is used for and how it should work.

---

persist(*df*, *lazy=False*, ***kwargs*)

Force materializing and caching the dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe
- **lazy** (*bool*) – `True`: first usage of the output will trigger persisting to happen; `False` (eager): persist is forced to happend immediately. Default to `False`
- **args** – parameter to pass to the underlying persist implementation
- **kwargs** (*Any*) – parameter to pass to the underlying persist implementation

Returns the persisted dataframe

Return type *fugue_spark.dataframe.SparkDataFrame*

Note: `persist` can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

---

register(*df*, *name*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) –
- **name** (*str*) –

Return type *fugue_spark.dataframe.SparkDataFrame*

repartition(*df*, *partition_spec*)
Partition the input dataframe using `partition_spec`.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how you want to partition the dataframe

Returns repartitioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing please read the Partition Tutorial

---

sample(*df*, *n=None*, *frac=None*, *replace=False*, *seed=None*, *metadata=None*)
Sample dataframe by number of rows or by fraction

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*Optional[int]*) – number of rows to sample, one and only one of `n` and `frac` must be set
- **frac** (*Optional[float]*) – fraction [0,1] to sample, one and only one of `n` and `frac` must be set
- **replace** (*bool*) – whether replacement is allowed. With replacement, there may be duplicated rows in the result, defaults to False
- **seed** (*Optional[int]*) – seed for randomness, defaults to None
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns sampled dataframe

Return type *DataFrame*

save_df(*df*, *path*, *format_hint=None*, *mode='overwrite'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *force_single=False*, ***kwargs*)
Save dataframe to a persistent storage

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **path** (*str*) – output path
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how to partition the dataframe before saving, defaults to empty
- **force_single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Load & Save.

property spark_session: pyspark.sql.session.SparkSession

Returns The wrapped spark session

Return type pyspark.sql.SparkSession

subtract(*df1*, *df2*, *distinct=True*, *metadata=None*)
`df1 - df2`

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `EXCEPT` (== `EXCEPT DISTINCT`), `false` for `EXCEPT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

take(*df*, *n*, *presort*, *na_position='last'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *metadata=None*)
Get the first n rows of a DataFrame per partition. If a presort is defined, use the presort before applying take. presort overrides partition_spec.presort. The Fugue implementation of the presort follows Pandas convention of specifying NULLs first or NULLs last. This is different from the Spark and SQL convention of NULLs as the smallest value.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*int*) – number of rows to return
- **presort** (*str*) – presort expression similar to partition presort
- **na_position** (*str*) – position of null values during the presort. can accept `first` or `last`
- **partition_spec** (fugue.collections.partition.PartitionSpec) – PartitionSpec to apply the take operation
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns n rows of DataFrame per partition

Return type *DataFrame*

to_df(*df*, *schema=None*, *metadata=None*)
Convert a data structure to *SparkDataFrame*

Parameters

- **data** – *DataFrame*, pyspark.sql.DataFrame, pyspark.RDD, pandas DataFrame or list or iterable of arrays

- **schema** (*Optional[Any]*) – Schema like object or pyspark.sql.types.StructType defaults to None.
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None
- **df** (*Any*) –

Returns engine compatible dataframe

Return type *fugue_spark.dataframe.SparkDataFrame*

Note:

- if the input is already *SparkDataFrame*, it should return itself
- For RDD, list or iterable of arrays, `schema` must be specified
- When `schema` is not None, a potential type cast may happen to ensure the dataframe's schema.
- all other methods in the engine can take arbitrary dataframes and call this method to convert before doing anything

---

union(*df1*, *df2*, *distinct=True*, *metadata=None*)

Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `UNION` (== `UNION DISTINCT`), `false` for `UNION ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

class fugue_spark.execution_engine.**SparkSQLEngine**(*execution_engine*)

Bases: *fugue.execution.execution_engine.SQLEngine*

Spark SQL execution implementation.

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) – it must be *SparkExecutionEngine*

Raises **ValueError** – if the engine is not *SparkExecutionEngine*

select(*dfs*, *statement*)

Execute select statement on the sql engine.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **statement** (*str*) – the `SELECT` statement using the `dfs` keys as tables

Returns result of the `SELECT` statement

Return type *fugue.dataframe.dataframe.DataFrame*

Examples

```
>>> dfs = DataFrames(a=df1, b=df2)
>>> sql_engine.select(dfs, "SELECT * FROM a UNION SELECT * FROM b")
```

Note: There can be tables that is not in `dfs`. For example you want to select from hive without input DataFrames:

```
>>> sql_engine.select(DataFrames(), "SELECT * FROM hive.a.table")
```

4.4.3 fugue_spark.ibis_engine

class fugue_spark.ibis_engine.**SparkIbisEngine**(*execution_engine*)

Bases: *fugue_ibis.execution.ibis_engine.IbisEngine*

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) –

Return type None

select(*dfs*, *ibis_func*)

Execute the ibis select expression.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **ibis_func** (*Callable[[ibis.backends.base.BaseBackend], ibis.expr.types.TableExpr]*) – the ibis compute function

Returns result of the ibis function

Return type *fugue.dataframe.dataframe.DataFrame*

Note: This interface is experimental, so it is subjected to change.

4.4.4 fugue_spark.registry

4.5 fugue_dask

4.5.1 fugue_dask.dataframe

class fugue_dask.dataframe.**DaskDataFrame**(*df=None*, *schema=None*, *metadata=None*, *num_partitions=0*, *type_safe=True*)

Bases: *fugue.dataframe.dataframe.DataFrame*

DataFrame that wraps Dask DataFrame. Please also read the DataFrame Tutorial to understand this Fugue concept

Parameters

- **df** (*Any*) – dask.dataframe.DataFrame, pandas DataFrame or list or iterable of arrays
- **schema** (*Any*) – Schema like object or pyspark.sql.types.StructType, defaults to None.
- **metadata** (*Any*) – Parameters like object, defaults to None
- **num_partitions** (*int*) – initial number of partitions for the dask dataframe defaults to 0 to get the value from *fugue.dask.dataframe.default.partitions*
- **type_safe** – whether to cast input data to ensure type safe, defaults to True

---

Note: For dask.dataframe.DataFrame, schema must be None

---

alter_columns(*columns*)
Change column types

Parameters **columns** (*Any*) – Schema like object, all columns should be contained by the dataframe schema

Returns a new dataframe with altered columns, the order of the original schema will not change

Return type *fugue.dataframe.dataframe.DataFrame*

as_array(*columns=None*, *type_safe=False*)
Convert to 2-dimensional native python array

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns 2-dimensional native python array

Return type List[Any]

Note: If type_safe is False, then the returned values are 'raw' values.

---

as_array_iterable(*columns=None*, *type_safe=False*)
Convert to iterable of native python arrays

Parameters

- **columns** (*Optional[List[str]]*) – columns to extract, defaults to None
- **type_safe** (*bool*) – whether to ensure output conforms with its schema, defaults to False

Returns iterable of native python arrays

Return type Iterable[Any]

Note: If type_safe is False, then the returned values are 'raw' values.

---

as_local()
Convert this dataframe to a *LocalDataFrame*

Return type *fugue.dataframe.dataframe.LocalDataFrame*

as_pandas()
Convert to pandas DataFrame

Return type pandas.core.frame.DataFrame

count()
Get number of rows of this dataframe

Return type int

property empty: bool
Whether this dataframe is empty

head(*n*, *columns=None*)
Get first n rows of the dataframe as 2-dimensional array :param n: number of rows :param columns: selected columns, defaults to None (all columns) :return: 2-dimensional array

Parameters

- **n** (*int*) –
- **columns** (*Optional[List[str]]*) –

Return type List[Any]

property is_bounded: bool
Whether this dataframe is bounded

property is_local: bool
Whether this dataframe is a *LocalDataFrame*

property native: dask.dataframe.core.DataFrame
The wrapped Dask DataFrame

Return type dask.dataframe.DataFrame

property num_partitions: int
Number of physical partitions of this dataframe. Please read the Partition Tutorial

peek_array()
Peek the first row of the dataframe as array

Raises *FugueDataFrameEmptyError* – if it is empty

Return type Any

persist(***kwargs*)

Parameters **kwargs** (*Any*) –

Return type *fugue_dask.dataframe.DaskDataFrame*

rename(*columns*)
Rename the dataframe using a mapping dict

Parameters **columns** (*Dict[str, str]*) – key: the original column name, value: the new name

Returns a new dataframe with the new names

Return type *fugue.dataframe.dataframe.DataFrame*

4.5.2 fugue_dask.execution_engine

class fugue_dask.execution_engine.**DaskExecutionEngine**(*dask_client=None*, *conf=None*)
Bases: *fugue.execution.execution_engine.ExecutionEngine*

The execution engine based on Dask.

Please read the ExecutionEngine Tutorial to understand this important Fugue concept

Parameters

- **dask_client** (*Optional[distributed.client.Client]*) – Dask distributed client, defaults to None. If None, then it will try to get the current active global client. If there is no active client, it will create and use a global *Client(processes=True)*
- **conf** (*Any*) – Parameters like object defaults to None, read the Fugue Configuration Tutorial to learn Fugue specific options

---

Note: You should setup Dask single machine or distributed environment in the common way. Before initializing *DaskExecutionEngine*

---

broadcast(*df*)
Broadcast the dataframe to all workers for a distributed computing framework

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe

Returns the broadcasted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

property dask_client: distributed.client.Client
The Dask Client associated with this engine

property default_sql_engine: *fugue.execution.execution_engine.SQLEngine*
Default SQLEngine if user doesn't specify

distinct(*df*, *metadata=None*)
Equivalent to `SELECT DISTINCT * FROM df`

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – dataframe
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns [description]

Return type *DataFrame*

dropna(*df*, *how='any'*, *thresh=None*, *subset=None*, *metadata=None*)
Drop NA recods from dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **how** (*str*) – 'any' or 'all'. 'any' drops rows that contain any nulls. 'all' drops rows that contain all nulls.
- **thresh** (*Optional[int]*) – int, drops rows that have less than thresh non-null values
- **subset** (*Optional[List[str]]*) – list of columns to operate on

- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records dropped

Return type *DataFrame*

fillna(*df*, *value*, *subset=None*, *metadata=None*)
Fill NULL, NAN, NAT values in a dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **value** (*Any*) – if scalar, fills all columns with same value. if dictionary, fills NA using the keys as column names and the values as the replacement values.
- **subset** (*Optional[List[str]]*) – list of columns to operate on. ignored if value is a dictionary
- **metadata** (*Any, optional*) – dict-like object to add to the result dataframe, defaults to None

Returns DataFrame with NA records filled

Return type *DataFrame*

property fs: triad.collections.fs.FileSystem
File system of this engine instance

intersect(*df1*, *df2*, *distinct=True*, *metadata=None*)
Intersect df1 and df2

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – true for INTERSECT (== INTERSECT DISTINCT), false for INTERSECT ALL
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of df1 and df2 must be identical, or an exception will be thrown.

---

join(*df1*, *df2*, *how*, *on=[]*, *metadata=None*)
Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **how** (*str*) – can accept semi, left_semi, anti, left_anti, inner, left_outer, right_outer, full_outer, cross
- **on** (*List[str]*) – it can always be inferred, but if you provide, it will be validated against the inferred keys.

- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the joined dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Please read this

---

load_df(*path*, *format_hint=None*, *columns=None*, ***kwargs*)
Load dataframe from persistent storage

Parameters

- **path** (*Union[str, List[str]]*) – the path to the dataframe
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **columns** (*Optional[Any]*) – list of columns or a Schema like object, defaults to None
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Returns an engine compatible dataframe

Return type *fugue_dask.dataframe.DaskDataFrame*

For more details and examples, read Zip & Comap.

property log: logging.Logger
Logger of this engine instance

map(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)
Apply a function to each partition after you partition the data in a specified way.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframe.LocalDataFrame*]*, fugue.dataframe.dataframe.LocalDataFrame*]*) – the function to apply on every logical partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframe.DataFrame*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the map operation

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing, you must read this to understand what map is used for and how it should work.

---

persist(*df*, *lazy=False*, ***kwargs*)
Force materializing and caching the dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe
- **lazy** (*bool*) – True: first usage of the output will trigger persisting to happen; False (eager): persist is forced to happend immediately. Default to False
- **args** – parameter to pass to the underlying persist implementation
- **kwargs** (*Any*) – parameter to pass to the underlying persist implementation

Returns the persisted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: persist can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

---

property pl_utils: fugue_dask._utils.DaskUtils
Pandas-like dataframe utils

repartition(*df*, *partition_spec*)
Partition the input dataframe using partition_spec.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how you want to partition the dataframe

Returns repartitioned dataframe

Return type *fugue_dask.dataframe.DaskDataFrame*

Note: Before implementing please read the Partition Tutorial

---

sample(*df*, *n=None*, *frac=None*, *replace=False*, *seed=None*, *metadata=None*)
Sample dataframe by number of rows or by fraction

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*Optional[int]*) – number of rows to sample, one and only one of n and frac must be set
- **frac** (*Optional[float]*) – fraction [0,1] to sample, one and only one of n and frac must be set
- **replace** (*bool*) – whether replacement is allowed. With replacement, there may be duplicated rows in the result, defaults to False
- **seed** (*Optional[int]*) – seed for randomness, defaults to None
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns sampled dataframe

Return type *DataFrame*

save_df(*df*, *path*, *format_hint=None*, *mode='overwrite'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *force_single=False*, ***kwargs*)

Save dataframe to a persistent storage

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **path** (*str*) – output path
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how to partition the dataframe before saving, defaults to empty
- **force_single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Load & Save.

subtract(*df1*, *df2*, *distinct=True*, *metadata=None*)

`df1 - df2`

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `EXCEPT` (`== EXCEPT DISTINCT`), `false` for `EXCEPT ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

take(*df*, *n*, *presort*, *na_position='last'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *metadata=None*)

Get the first n rows of a DataFrame per partition. If a presort is defined, use the presort before applying take. presort overrides partition_spec.presort. The Fugue implementation of the presort follows Pandas convention of specifying NULLs first or NULLs last. This is different from the Spark and SQL convention of NULLs as the smallest value.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **n** (*int*) – number of rows to return
- **presort** (*str*) – presort expression similar to partition presort

- **na_position** (*str*) – position of null values during the presort. can accept `first` or `last`
- **partition_spec** (fugue.collections.partition.PartitionSpec) – PartitionSpec to apply the take operation
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns n rows of DataFrame per partition

Return type *DataFrame*

to_df(*df*, *schema=None*, *metadata=None*)

Convert a data structure to *DaskDataFrame*

Parameters

- **data** – *DataFrame*, dask.dataframe.DataFrame, pandas DataFrame or list or iterable of arrays
- **schema** (*Optional[Any]*) – Schema like object, defaults to None.
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None
- **df** (*Any*) –

Returns engine compatible dataframe

Return type *fugue_dask.dataframe.DaskDataFrame*

Note:

- if the input is already *DaskDataFrame*, it should return itself
- For list or iterable of arrays, `schema` must be specified
- When `schema` is not None, a potential type cast may happen to ensure the dataframe's schema.
- all other methods in the engine can take arbitrary dataframes and call this method to convert before doing anything

---

union(*df1*, *df2*, *distinct=True*, *metadata=None*)

Join two dataframes

Parameters

- **df1** (fugue.dataframe.dataframe.DataFrame) – the first dataframe
- **df2** (fugue.dataframe.dataframe.DataFrame) – the second dataframe
- **distinct** (*bool*) – `true` for `UNION` (== `UNION DISTINCT`), `false` for `UNION ALL`
- **metadata** (*Optional[Any]*) – dict-like object to add to the result dataframe, defaults to None

Returns the unioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Currently, the schema of `df1` and `df2` must be identical, or an exception will be thrown.

---

class fugue_dask.execution_engine.**QPDDaskEngine**(*execution_engine*)
Bases: *fugue.execution.execution_engine.SQLEngine*

QPD execution implementation.

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) – the execution engine this sql engine will run on

select(*dfs*, *statement*)
Execute select statement on the sql engine.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **statement** (*str*) – the SELECT statement using the dfs keys as tables

Returns result of the SELECT statement

Return type *fugue.dataframe.dataframe.DataFrame*

Examples

```
>>> dfs = DataFrames(a=df1, b=df2)
>>> sql_engine.select(dfs, "SELECT * FROM a UNION SELECT * FROM b")
```

Note: There can be tables that is not in dfs. For example you want to select from hive without input DataFrames:

```
>>> sql_engine.select(DataFrames(), "SELECT * FROM hive.a.table")
```

4.5.3 fugue_dask.ibis_engine

class fugue_dask.ibis_engine.**DaskIbisEngine**(*execution_engine*)
Bases: *fugue_ibis.execution.ibis_engine.IbisEngine*

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) –

Return type None

select(*dfs*, *ibis_func*)
Execute the ibis select expression.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **ibis_func** (*Callable[[ibis.backends.base.BaseBackend], ibis.expr.types.TableExpr]*) – the ibis compute function

Returns result of the ibis function

Return type *fugue.dataframe.dataframe.DataFrame*

Note: This interface is experimental, so it is subjected to change.

4.5.4 fugue_dask.registry

4.6 fugue_ray

4.6.1 fugue_ray.dateframe

4.6.2 fugue_ray.execution_engine

class fugue_ray.execution_engine.**RayExecutionEngine**(*conf=None*, *connection=None*)

Bases: *fugue_duckdb.execution_engine.DuckExecutionEngine*

A hybrid engine of Ray and DuckDB as Phase 1 of Fugue Ray integration. Most operations will be done by DuckDB, but for `map`, it will use Ray.

Parameters

- **conf** (*Any*) – Parameters like object, read the Fugue Configuration Tutorial to learn Fugue specific options
- **connection** (*Optional[duckdb.DuckDBPyConnection]*) – DuckDB connection

broadcast(*df*)

Broadcast the dataframe to all workers for a distributed computing framework

Parameters **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe

Returns the broadcasted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

convert_yield_dataframe(*df*, *as_local*)

Convert a yield dataframe to a dataframe that can be used after this execution engine stops.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – DataFrame
- **as_local** (*bool*) – whether yield a local dataframe

Returns another DataFrame that can be used after this execution engine stops

Return type *fugue.dataframe.dataframe.DataFrame*

Note: By default, the output dataframe is the input dataframe. But it should be overridden if when an engine stops and the input dataframe will become invalid.

For example, if you custom a spark engine where you start and stop the spark session in this engine's *start_engine()* and *stop_engine()*, then the spark dataframe will be invalid. So you may consider converting it to a local dataframe so it can still exist after the engine stops.

load_df(*path*, *format_hint=None*, *columns=None*, ***kwargs*)

Load dataframe from persistent storage

Parameters

- **path** (*Union[str, List[str]]*) – the path to the dataframe
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **columns** (*Optional[Any]*) – list of columns or a Schema like object, defaults to None
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Returns an engine compatible dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

For more details and examples, read Zip & Comap.

map(*df*, *map_func*, *output_schema*, *partition_spec*, *metadata=None*, *on_init=None*)

Apply a function to each partition after you partition the data in a specified way.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **map_func** (*Callable[[*fugue.collections.partition.PartitionCursor, fugue.dataframe.dataframe.LocalDataFrame*], *fugue.dataframe.dataframe.LocalDataFrame*]*) – the function to apply on every logical partition
- **output_schema** (*Any*) – Schema like object that can't be None. Please also understand why we need this
- **partition_spec** (fugue.collections.partition.PartitionSpec) – partition specification
- **metadata** (*Optional[Any]*) – dict-like metadata object to add to the dataframe after the map operation, defaults to None
- **on_init** (*Optional[Callable[[int,* fugue.dataframe.dataframe.DataFrame*], Any]]*) – callback function when the physical partition is initializaing, defaults to None

Returns the dataframe after the map operation

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing, you must read this to understand what map is used for and how it should work.

---

persist(*df*, *lazy=False*, ***kwargs*)

Force materializing and caching the dataframe

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – the input dataframe
- **lazy** (*bool*) – `True`: first usage of the output will trigger persisting to happen; `False` (eager): persist is forced to happend immediately. Default to `False`
- **args** – parameter to pass to the underlying persist implementation
- **kwargs** (*Any*) – parameter to pass to the underlying persist implementation

Returns the persisted dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: `persist` can only guarantee the persisted dataframe will be computed for only once. However this doesn't mean the backend really breaks up the execution dependency at the persisting point. Commonly, it doesn't cause any issue, but if your execution graph is long, it may cause expected problems for example, stack overflow.

repartition(*df*, *partition_spec*)
Partition the input dataframe using `partition_spec`.

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how you want to partition the dataframe

Returns repartitioned dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: Before implementing please read the Partition Tutorial

save_df(*df*, *path*, *format_hint=None*, *mode='overwrite'*, *partition_spec=PartitionSpec(num='0', by=[], presort='')*, *force_single=False*, ***kwargs*)
Save dataframe to a persistent storage

Parameters

- **df** (fugue.dataframe.dataframe.DataFrame) – input dataframe
- **path** (*str*) – output path
- **format_hint** (*Optional[Any]*) – can accept `parquet`, `csv`, `json`, defaults to None, meaning to infer
- **mode** (*str*) – can accept `overwrite`, `append`, `error`, defaults to "overwrite"
- **partition_spec** (fugue.collections.partition.PartitionSpec) – how to partition the dataframe before saving, defaults to empty
- **force_single** (*bool*) – force the output as a single file, defaults to False
- **kwargs** (*Any*) – parameters to pass to the underlying framework

Return type None

For more details and examples, read Load & Save.

to_df(*df*, *schema=None*, *metadata=None*)
Convert a data structure to this engine compatible DataFrame

Parameters

- **data** – *DataFrame*, pandas DataFramme or list or iterable of arrays or others that is supported by certain engine implementation
- **schema** (*Optional[Any]*) – Schema like object, defaults to None
- **metadata** (*Optional[Any]*) – Parameters like object, defaults to None
- **df** (*Any*) –

Returns engine compatible dataframe

Return type *fugue.dataframe.dataframe.DataFrame*

Note: There are certain conventions to follow for a new implementation:

- if the input is already in compatible dataframe type, it should return itself
- all other methods in the engine interface should take arbitrary dataframes and call this method to convert before doing anything

---

4.6.3 fugue_ray.registry

4.7 fugue_ibis

4.7.1 fugue_ibis.execution

fugue_ibis.execution.ibis_engine

class fugue_ibis.execution.ibis_engine.**IbisEngine**(*execution_engine*)

Bases: object

The abstract base class for different ibis execution implementations.

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) – the execution engine this ibis engine will run on

Return type None

property **execution_engine**: *fugue.execution.execution_engine.ExecutionEngine*

the execution engine this ibis engine will run on

abstract **select**(*dfs*, *ibis_func*)

Execute the ibis select expression.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **ibis_func** (*Callable[[ibis.backends.base.BaseBackend], ibis.expr.types.TableExpr]*) – the ibis compute function

Returns result of the ibis function

Return type *fugue.dataframe.dataframe.DataFrame*

Note: This interface is experimental, so it is subjected to change.

---

fugue_ibis.execution.ibis_engine.**register_ibis_engine**(*priority*, *func*)

Parameters

- **priority** (*int*) –
- **func** (*Callable[[fugue.execution.execution_engine.ExecutionEngine, Any], Optional[fugue_ibis.execution.ibis_engine.IbisEngine]]*) –

Return type None

fugue_ibis.execution.ibis_engine.**to_ibis_engine**(*execution_engine*, *ibis_engine=None*)

Parameters

- **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) –
- **ibis_engine** (*Optional[Any]*) –

Return type *fugue_ibis.execution.ibis_engine.IbisEngine*

fugue_ibis.execution.pandas_backend

class fugue_ibis.execution.pandas_backend.**PandasIbisEngine**(*execution_engine*)

Bases: *fugue_ibis.execution.ibis_engine.IbisEngine*

Parameters **execution_engine** (fugue.execution.execution_engine.ExecutionEngine) –

Return type None

select(*dfs*, *ibis_func*)

Execute the ibis select expression.

Parameters

- **dfs** (fugue.dataframe.dataframes.DataFrames) – a collection of dataframes that must have keys
- **ibis_func** (*Callable[[ibis.backends.base.BaseBackend], ibis.expr.types.TableExpr]*) – the ibis compute function

Returns result of the ibis function

Return type *fugue.dataframe.dataframe.DataFrame*

Note: This interface is experimental, so it is subjected to change.

4.7.2 fugue_ibis.extensions

fugue_ibis.extensions.**as_fugue**(*expr*, *ibis_engine=None*)

Convert a lazy ibis object to Fugue workflow dataframe

Parameters

- **expr** (*ibis.expr.types.TableExpr*) – the actual instance should be LazyIbisObject
- **ibis_engine** (*Optional[Any]*) –

Returns the Fugue workflow dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Examples

```
# non-magical approach
import fugue as FugueWorkflow
from fugue_ibis import as_ibis, as_fugue

dag = FugueWorkflow()
df1 = dag.df([[0]], "a:int")
df2 = dag.df([[1]], "a:int")
idf1 = as_ibis(df1)
idf2 = as_ibis(df2)
idf3 = idf1.union(idf2)
result = idf3.mutate(b=idf3.a+1)
as_fugue(result).show()
```

```
# magical approach
import fugue as FugueWorkflow
import fugue_ibis  # must import

dag = FugueWorkflow()
idf1 = dag.df([[0]], "a:int").as_ibis()
idf2 = dag.df([[1]], "a:int").as_ibis()
idf3 = idf1.union(idf2)
result = idf3.mutate(b=idf3.a+1).as_fugue()
result.show()
```

Note: The magic is that when importing `fugue_ibis`, the functions `as_ibis` and `as_fugue` are added to the correspondent classes so you can use them as if they are parts of the original classes.

This is an idea similar to patching. Ibis uses this programming model a lot. Fugue provides this as an option.

Note: The returned object is not really a `TableExpr`, it's a 'super lazy' object that will be translated into `TableExpr` at run time. This is because to compile an ibis execution graph, the input schemas must be known. However, in Fugue, this is not always true. For example if the previous step is to pivot a table, then the output schema can be known at runtime. So in order to be a part of Fugue, we need to be able to construct ibis expressions before knowing the input schemas.

---

fugue_ibis.extensions.**as_ibis**(*df*)

Convert the Fugue workflow dataframe to an ibis table for ibis operations.

Parameters **df** (fugue.workflow.workflow.WorkflowDataFrame) – the Fugue workflow dataframe

Returns the object representing the ibis table

Return type ibis.expr.types.TableExpr

Examples

```
# non-magical approach
import fugue as FugueWorkflow
from fugue_ibis import as_ibis, as_fugue
```

(continues on next page)

(continued from previous page)

```
dag = FugueWorkflow()
df1 = dag.df([[0]], "a:int")
df2 = dag.df([[1]], "a:int")
idf1 = as_ibis(df1)
idf2 = as_ibis(df2)
idf3 = idf1.union(idf2)
result = idf3.mutate(b=idf3.a+1)
as_fugue(result).show()
```

```
# magical approach
import fugue as FugueWorkflow
import fugue_ibis  # must import

dag = FugueWorkflow()
idf1 = dag.df([[0]], "a:int").as_ibis()
idf2 = dag.df([[1]], "a:int").as_ibis()
idf3 = idf1.union(idf2)
result = idf3.mutate(b=idf3.a+1).as_fugue()
result.show()
```

Note: The magic is that when importing `fugue_ibis`, the functions `as_ibis` and `as_fugue` are added to the correspondent classes so you can use them as if they are parts of the original classes.

This is an idea similar to patching. Ibis uses this programming model a lot. Fugue provides this as an option.

Note: The returned object is not really a `TableExpr`, it's a 'super lazy' object that will be translated into `TableExpr` at run time. This is because to compile an ibis execution graph, the input schemas must be known. However, in Fugue, this is not always true. For example if the previous step is to pivot a table, then the output schema can be known at runtime. So in order to be a part of Fugue, we need to be able to construct ibis expressions before knowing the input schemas.

`fugue_ibis.extensions.`**`run_ibis`**(*ibis_func*, *ibis_engine=None*, ***dfs*)

Run an ibis workflow wrapped in `ibis_func`

Parameters

- **ibis_func** (*Callable[[ibis.backends.base.BaseBackend], ibis.expr.types.TableExpr]*) – the function taking in an ibis backend, and returning an Ibis TableExpr
- **ibis_engine** (*Optional[Any]*) – an object that together with *ExecutionEngine* can determine *IbisEngine* , defaults to None
- **dfs** (`fugue.workflow.workflow.WorkflowDataFrame`) – dataframes in the same workflow

Returns the output workflow dataframe

Return type *fugue.workflow.workflow.WorkflowDataFrame*

Examples

```
import fugue as FugueWorkflow
from fugue_ibis import run_ibis

def func(backend):
    t = backend.table("tb")
    return t.mutate(b=t.a+1)

dag = FugueWorkflow()
df = dag.df([[0]], "a:int")
result = run_ibis(func, tb=df)
result.show()
```

PYTHON MODULE INDEX

f

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

S

T

U

V

W

Y

Z