FLIP-38: Python Table API

Discussion thread	
Vote thread	
JIRA	~~FLINK-12308~~ - *Support python language in Flink Table API* CLOSED
Release	1.10

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

At the Flink API level, we have DataStreamAPI/DataSetAPI/TableAPI&SQL, the Table API will become the first-class citizen. Table API is declarative and can be automatically optimized, which is mentioned in the Flink mid-term roadmap by Stephan. So, first considering supporting Python at the Table level to cater to the current large number of analytics users. So this proposal will cover the following items:

- Python TableAPI Interface

Introduces a set of Python Table API which should mirror Java / Scala Table API, i.e. the interfaces should including such as Table, TableEnvironment, TableConfig, etc.

- Python TableAPI Implementation Architecture

We will offer two alternative architecture options, one for pure Python language support and one for extended multi-language design(long-term goals).

- Job Submission

Python Table API programs should be similarly submitted and deployed as Java / Scala Table API programs, Such as CLI, web, containerized, etc.

Modules

- flink-python(maven module)
  - pyflink(python package)
    - table
    - shell
    - streaming(in the future)
    - others...
- flink-clients(maven module)

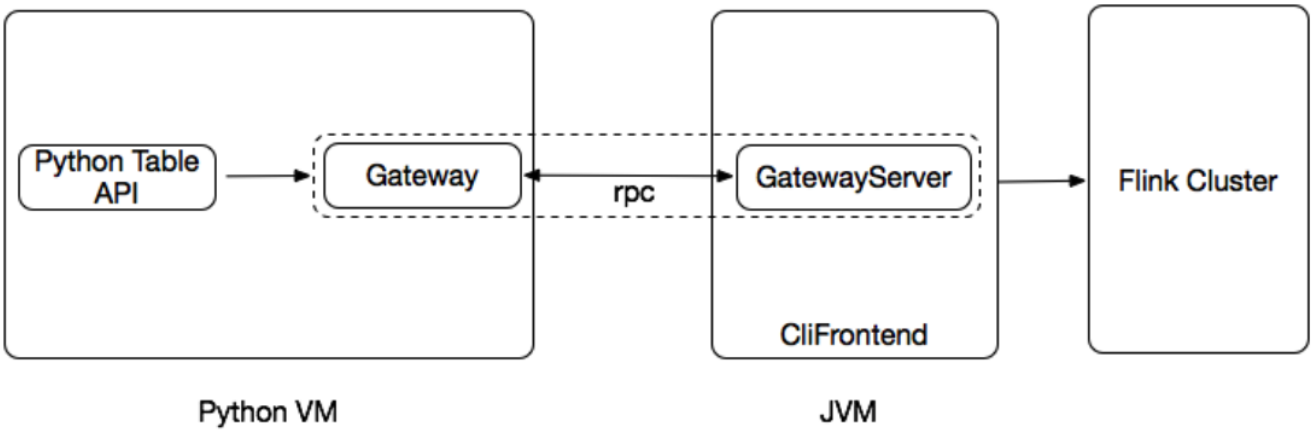
Support for submitting Python Table API job in CliFrontend, such as `flink run -py wordcount.py`.

We need to add components in FLINK JIRA as follows:

- API/Python - for Python API (already exists)
- Python Shell - for interactive Python program

Architecture

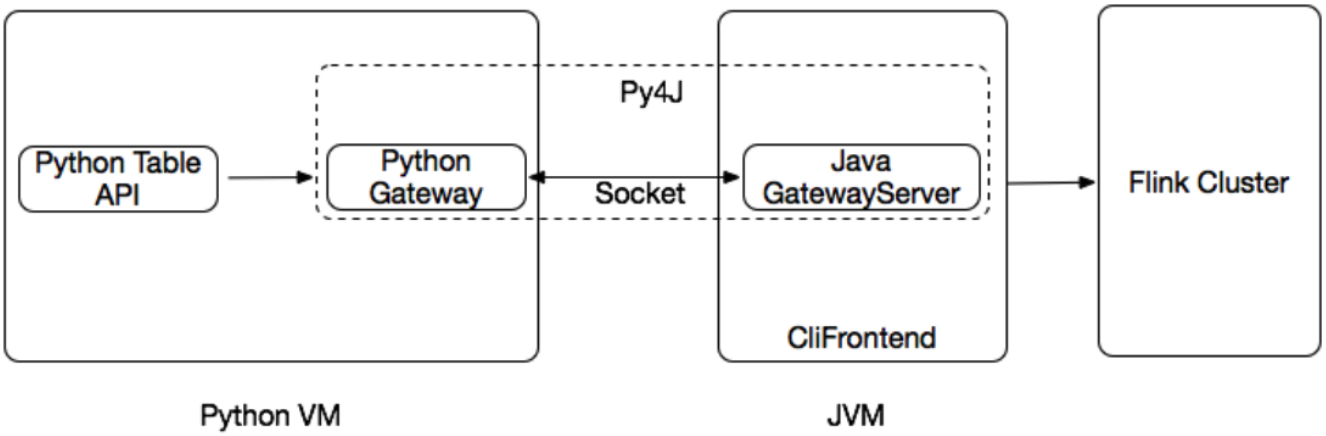
We don't develop python operators like `flink-python` and `flink-stream-python`. To get the most out of the existing Java/Scala results (the Calcite-based optimizer), the Python Table API only needs to define the Python Table API interface. Calls to the existing Java Table API implementation to meet the needs of python users with minimal effort. So our main job is to implement communication between Python VM and Java VM, as shown below:



Currently, we have two options for the implementation of RPC.

Approach 1

For Python API, a very mature solution is to choose Py4J as shown below:



At Python Side, Py4J provides a JavaGateway object. It has a field “jvm” which enables Python program to access the Java classes directly. We can construct the Python wrappers for the Java classes through it.

At the Java side, Py4J provides GatewayServer. It receives the Python API requests and we can use it to delegate all the method calling of the Python API to the corresponding Java/Scala Table API.

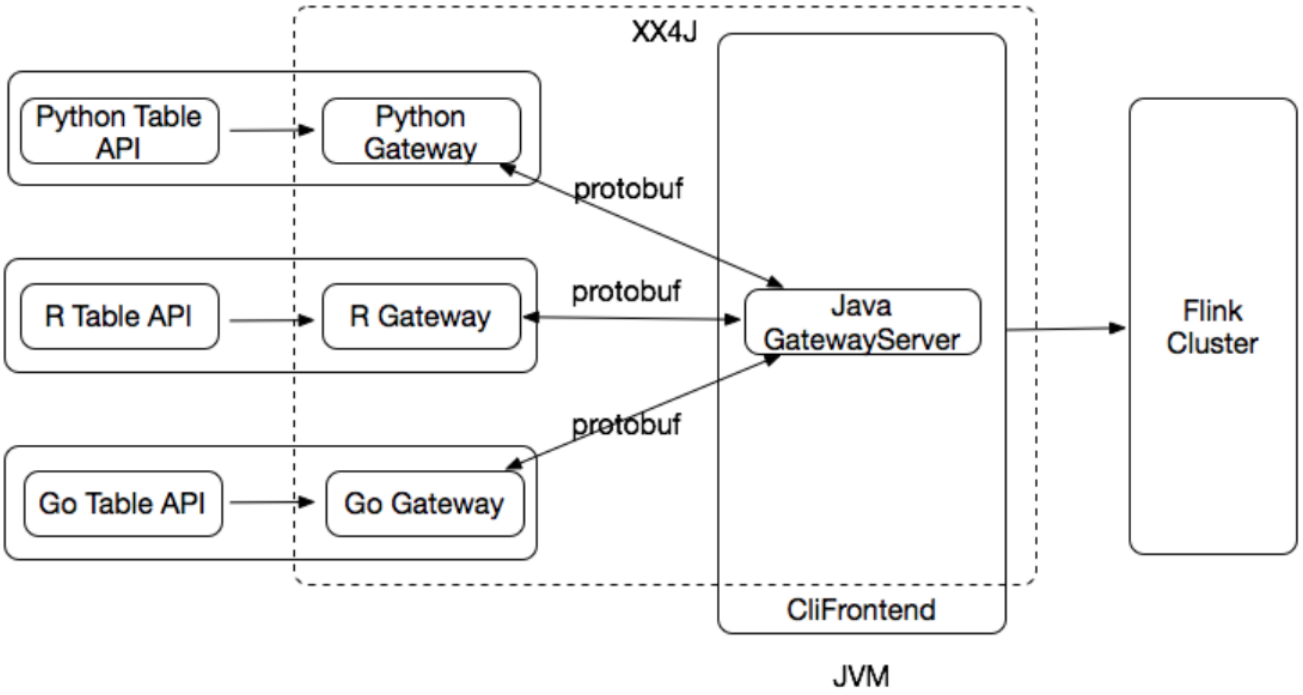
There will be Python wrappers for all the API classes such as TableEnvironment, Table, TableSink, TableSource, Catalog, etc. For example:

```
class Table(object):
    """
    Wrapper of org.apache.flink.table.api.Table
    """
    def __init__(self, j_table):
        self._j_table = j_table  # The references to table object on JVM.
    def select(self, col_list):
        return Table(self._j_table.select(col_list))  # Call the java table api method directly
…
...
```

In this approach Py4J encapsulates all serialization deserialization processes.

Approach 2

Currently, we want to add Python Table API, And in the future, we may want to support the other popular Non-JVM language based Table API, such as R, Go, etc. So we can also have a more scalable approach, as follows:



In this approach, we can replace Py4J with a similar component(suppose named XX4J) which can support all kinds of languages besides Python as the bridge between the client language Table API and Java Table API. From the architecture graph above, we can see that a language-neutral transportation layer (such as protobuf) makes it easy to support new kinds of languages as the Table API.

But by now, Python and Java communication mechanism undoubtedly choose Py4J is the most mature and lowest cost solution. I also do a POC using Py4J**. So for the API level, we make the following plan**

- **The short-term:**

We may initially go with a simple approach to map the Python Table API to the Java Table API via Py4J.

- **The long-term:**

We may need to create a Python API that follows the same structure as Flink's Table API that produces the language-independent DAG. (As Stephan already motioned on the mailing thread)

Public Interfaces

The Python Table API classes are just some wrappers of the corresponding Java Table API classes. All the user interface in `flink-table-common` and `flink-table-api-java` should be defined in Python Table API. The main interface/implementation is as follows:

- Table
- TableEnvironment
- TableConfig
- Descriptor
- TypeInformation(May change after FLIP-37)
- etc

Table

```
class Table(object):
    """
    Wrapper of org.apache.flink.table.api.Table
    """
    def __init__(self, j_table):
        self._j_table = j_table  # The references to table object on JVM.
    def select(self, col_list):
        return Table(self._j_table.select(col_list))  # Call the java table api method directly
    def where(self, condition):
        return Table(self._j_table.where(condition))
    …
    …
    def write_to_sink(self, j_sink):
        self._j_table.writeToSink(j_sink._j_table_sink)
     …
    ....
```

GroupedTable

```
class GroupedTable(object):
    """
    Wrapper of org.apache.flink.table.api.GroupedTable
    """
    def __init__(self, j_grouped_table):
        # The references to group table object on JVM.
        self._j_grouped_table = j_grouped_table
    def select(self, col_list):
        # Call the java table api method directly
        return Table(self._j_grouped_table.select
(col_list))
```

GroupWindowedTable, WindowGroupedTable, OverWindowedTable are defined similarly as GroupTable.

TableConfig

```
class TableConfig(object):
    def __init__(self, j_table_config=None):
            gateway = get_gateway()
            if j_table_config is None:
                 self._j_table_config = gateway.jvm.TableConfig()
            else:
                self._j_table_config = j_table_config
  def get_local_timezone(self):
      return self._j_table_config.getLocalTimeZone().getId()

  def set_local_timezone(self, timezone_id):
   ...
def get_configuration(self):
       return Configuration(j_configuration=self._j_table_config.getConfiguration())
def add_configuration(self, configuration):
      self._j_table_config.addConfiguration(configuration._j_configuration)
```

TableEnvironment

```
class TableEnvironment(object):
  """
  Wrap and extend for org.apache.flink.table.api.TableEnvironment
  """
  table_config = None
  def __init__(self, j_tenv):
      self._j_tenv = j_tenv
  def register_table(self, name, table):
      self._j_tenv.registerTable(name, table._j_table)
  def register_table_source(self, name, table_source):
      self._j_tenv.registerTableSource(name, table_source.j_table_source)
  def register_table_sink(self, name, table_sink):
      self._j_tenv.registerTableSink(name, table_sink.j_table_sink)
  def scan(self, *table_path):
      j_paths = TypesUtil.convert_py_list_to_j_array("java.lang.String", table_path)
      j_table = self._j_tenv.scan(j_paths)
      return Table(j_table)
  def connect(self, connector_descriptor):
      return TableDescriptor(self._j_tenv.connect(connector_descriptor._j_connector_descriptor))
   …
   …
  def sql_query(self, query):
      j_table = self._j_tenv.sqlQuery(query)
      return Table(j_table)
```

```
def sql_update(self, stmt, config=None):
    if config is not None:
        j_table = self._j_tenv.sqlUpdate(stmt, config)
    else:
        j_table = self._j_tenv.sqlUpdate(stmt)
# Extension methods
def from_collection(self, data):
    …
    return Table(...)


def execute(self):
    self._j_tenv.execEnv().execute()
def set_parallelism(self, parallelism):
    self._j_tenv.execEnv().setParallelism(parallelism)
…
...
@classmethod
def create(cls, table_config):
    j_tenv = ...
    return TableEnvironment(j_tenv)
```

Descriptor

There are a lot of Descriptor related classes, we will take Csv as an example:

```
class Csv(FormatDescriptor):
  def __init__(self):
      self._j_csv = _jvm.org.apache.flink.table.descriptors.Csv()
      super(Csv, self).__init__(self._j_csv)
  def field_delimiter(self, delimiter):
      self._j_csv.fieldDelimiter(delimiter)
      return self
  def line_delimiter(self, delimiter):
      self._j_csv.lineDelimiter(delimiter)
      return self
  def quote_character(self, quoteCharacter):
      self._j_csv.quoteCharacter(quoteCharacter)
      return self
  def allow_comments(self):
      self._j_csv.allowComments()
      return self
  def ignore_parse_errors(self):
      self._j_csv.ignoreParseErrors()
      return self
  def array_element_delimiter(self, delimiter):
      self._j_csv.arrayElementDelimiter(delimiter)
      return self
  def escape_character(self, escape_character):
      self._j_csv.escapeCharacter(escape_character)
      return self
  def null_literal(self, null_literal):
      self._j_csv.nullLiteral(null_literal)
      return self
```

Expression

Expression API will be not supported in the initial version as there is an ongoing work of introducing Table API Java Expression DSL. Python Expression API will leverage that and supported once that work is done.

DIST

Create a python directory in the published opt directory. The contents of python are as follows:

opt/python

 README.MD

 lib

   py4j-x.y.z-src.zip

   py4j-LICENSE.txt

   pyflink.zip

...

The flink-python module will be packaged as pyflink.zip, And put it in to opt/python/lib directory with PY4J_LICENSE.txt py4j-xxx-src.zip.

- And the pyflink shell we be added in published bin directory.
- The shell of `flink` should add some options for Python Table API, such as:
    - -py --python
    - -pyfs --py-files
    - etc ...

The detail can be found in the Job Submission section.

Docs

- Add the description of `flink run -py xx.py` in CLI
- Add rest service API for submit job
- Add a Python REPL submenu under the Deployment & Operations directory to add documentation for the python shell.
- Add Python Table API doc in current TableAPI doc
- Add Common concepts doc for Python Table API, in Basic concepts doc
- Add pythondocs at the same level as javadocs and scaladocs
- etc.

Examples

WordCount

Let's take a word count as an example to provide a example. The Python Table API will look like the following :

```
…

from pyflink.dataset import ExecutionEnvironment
from pyflink.table import BatchTableEnvironment, TableConfig
...

content = "..."

t_config = TableConfig()
env = ExecutionEnvironment.get_execution_environment()
t_env = BatchTableEnvironment.create(env, t_config)

# register Results table in table environment
tmp_dir = tempfile.gettempdir()
result_path = tmp_dir + '/result'
if os.path.exists(result_path):
try:
    if os.path.isfile(result_path):
        os.remove(result_path)
    else:
        shutil.rmtree(result_path)
except OSError as e:
    logging.error("Error removing directory: %s - %s.", e.filename, e.strerror)

t_env.connect(FileSystem().path(result_path)) \
    .with_format(OldCsv()
    .field_delimiter(',')
    .field("word", DataTypes.STRING())
    .field("count", DataTypes.BIGINT())) \
    .with_schema(Schema()
    .field("word", DataTypes.STRING())
    .field("count", DataTypes.BIGINT())) \
    .register_table_sink("Results")

elements = [(word, 1) for word in content.split(" ")]
t_env.from_elements(elements, ["word", "count"]) \
    .group_by("word") \
    .select("word, count(1) as count") \
    .insert_into("Results")

t_env.execute("word_count")

...
```

Job Submission

Flink Run

Support for submitting Python Table API job in CliFrontendAnd using `flink run` submit Python Table API job. The current `flink` command command line syntax is as follows:

```
flink <ACTION> [OPTIONS] [ARGUMENTS]
```

On the basis of the current `run` ACTION, we add to Python Table API support, specific OPTIONS are as follows:

- -py --python <python-file-name>

Python script with the program entry point. We can configure dependent resources with the `--py-files` option.

- -pyfs --py-files <python-files>

Attach custom python files for job. Comma can be used as the separator to specify multiple files. The standard python resource file suffixes such as .py/.egg/.zip all also supported.

    - -pym --py-module <python-module> Python module with the program entry point. This option must be used in conjunction with ` --py-files`.
- -py-exec --py-exec <python-binary-path>

The Python binary to be used.

- -py-env --py-env <process/docker>

The execution environment: process or docker.

- -py-docker-image --py-docker-image <docker image>

The docker image to be used to run the python worker. Valid when the option of “-py-env” is docker.

NOTE: When options `py` and `pym` appear at the same time, option `py` will be valid.

We can submit the Python Table API jobs as follows:

```
// Submit a simple job without any dependencies and parameters

FLINK_HOME/bin/flink run -py example.py

// Submit a job with dependencies and parallelism 16

FLINK_HOME/bin/flink run -p 16 -py example.py -pyfs resources1.egg, resources2.zip, resource3.py

// Submit a job by configuring python class

FLINK_HOME/bin/flink run -pyfs resources1.egg, resources2.zip, resource3.py -pym org.apache.pyflink.example

// Submit a job with binary path and execution environment config

FLINK_HOME/bin/flink run -pyfs resources1.egg, resources2.zip, resource3.py -py-exec ./my_env/env/bin/python -py-env process -pym org.apache.pyflink.example

// Sumit a simple job in which the Python worker runs in a docker container

FLINK_HOME/bin/flink run --py-env docker --py-docker-image xxx -py example.py
```

Regarding the [ARGUMENTS] section of syntax, we reuse existing features without any extensions.

Python Shell

The main goal of Flink Python Shell is to provide an interactive way for users to write and execute flink Python Table API jobs.

By default, jobs are executed in a mini cluster (for beginner users to learn and research). If users want to submit jobs to a cluster, they need to configure additional parameters, such as the execution mode or the JobMaster address (standalone cluster), just like Flink Scala Shell.

The Python shell is responsible for confirming the location of the flink files, loading required dependencies and then executing the shell initialization logic.

The initialization procedure will import all the Flink Python Table API classes which should be exposed to the user.

Example:

```
…

...

>>>exec_env = ExecutionEnvironment.get_execution_environment()

>>>exec_env.set_parallelism(1)

>>>t_config = TableConfig()

>>>t_env =BatchTableEnvironment.create(t_config)

>>>data = [(1L, "Sunny"), (2L, "Eden")]

>>>source = t_env.from_collection(data, "id, name")

>>>source.select("id, name").insertInto(“print”)

>>>t_env.execute()

1,Sunny

2,Eden
```

REST API

Currently we have a complete set of REST APIs that can be used to submit Java jobs(https://ci.apache.org/projects/flink/flink-docs-release-1.8/monitoring/rest_api.html). We need to make some changes to the current REST API to support submitting Python Table API jobs:

Current REST API	New REST API	Remarks
GET /jars	GET /artifacts	Now it lists both jar files and python package files.
POST /jars /upload	POST /artifacts/upload	Allow users to upload both jar files and python package files. Use "Content-Type" in the request header to mark the file type.
DELETE /jars/: jarid	DELETE /artifacts/: artifactid	Delete files according to “artifactid”.
GET /jars/:jarid /plan	GET /artifacts/:artifactid /plan	We will introduce a new optional parameter: pythonEntryModule. It allows specifying a python entry point module other than __main__.py.
POST /jars/:jarid /run	POST /artifacts/: artifactid/run	We also introduce the optional pythonEntryModule parameter here, as same as above.

We can firstly deprecated the old REST APIs and remove them in one or two releases.

Proposed Changes

This FLIP proposes are Flink Table's new support for the Python language. We will add a Python-related module, Introduces a set of Python Table API which should mirror Java / Scala Table API, and Job Submission, etc. And we will add checks for all of the changes.

Compatibility, Deprecation, and Migration Plan

This FLIP proposes new functionality and operators for the Python Table API. The behavior of existing operators is not modified.

Test Plan

This FLIP proposes can check by both It test case and validate the test case.

Implementation plan

Flink-1.9.0 Work items include

1. Add the flink-python module and a submodule flink-python-table to Py4j dependency configuration and Scan, Projection, and Filter operator of the Python Table API, and can be run in IDE(with simple test).
2. Add a basic test framework, just like the existing Java TableAPI, abstract some TestBase.
3. Add integrated Tox for ensuring compatibility with the python2/3 version
4. Add simplicity support for submitting Python Table API job in CliFrontend, i.e. `flink run -py wordcount.py` can be work(with simple test).
5. Add all the TypeInformation support in Python Table API.
6. Add all of the existing interfaces in Java Table API such as Table, TableEnvironment, TableConfig, and Descriptor etc.
7. Add all the Source/Sink Descriptor support in Python Table API.
8. Add Pip support, Flink Python Table API can be installed in python repo.
9. Add Python Table API doc in current TableAPI doc
10. Add Common concepts doc for Python Table API, in Basic concepts doc
11. Add pythondocs at the same level as javadocs and scaladocs
12. Add convenience interfaces to Python TableEnvironment, such as `from_collection`, `set_parallelism`, etc. which from `StreamExecutionEnvironment`.
13. Add the description of `flink run -py xx.py` in CLI
14. Add a Python shell, expose the TableEnvironment to the user, and automatically import the common dependencies.
15. Add a Python REPL submenu under the Deployment & Operations directory to add documentation for the python shell.
16. Add Support single job contains multiple python scripts, i.e. add `-pyfs --py-files <py-files>` (Attach custom python files for job. Use ',' to separate multiple files) for flink run `.
17. Other JIRAs that are constantly improving and enhanced, such as Catalog, DDL, Expression DSL, etc.

Future or next step

Adds User-defined Function support in Python Table API.

Rejected Alternatives

Regarding the API, we can adopt a scheme similar to the Beam community, that is, we may need to create a Python API that follows the same structure as Flink's Table API that produces the language-independent DAG. This approach needs to do a lot of cooperation with the beam community and it as a solution for the long-term goals.

NOTE: I remove the user-defined function part from this FLIP, Because the design of supporting the user-defined function should be a very complex. we need to describe it in a separate FLIP and bring up a community discussion separately.