developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Jul 28, 2021

HOME

Documentation Status

Prototyping of the SDP data access library - implementing the memory data models from the SDP architecture.

CHAPTER

ONE

INSTALLATION

You will require at least Python 3.6. Do the following:

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

CHAPTER

TWO

EXAMPLE

The prototype comes with a number of example programs.

2.1 Streamer

Simply streaming data in chunks from one process to another. First start an Apache Plasma store:

```
$ plasma_store -s /tmp/plasma -m 1000000000
/arrow/cpp/src/plasma/store.cc:1242: Allowing the Plasma store to use up to 1GB of␣
↪memory.
/arrow/cpp/src/plasma/store.cc:1269: Starting object store with directory /dev/shm␣
↪and huge page support disabled
```

Note that `plasma_store` is the binary installed by the `pyarrow` package. It is generally called `plasma-store-server` if installed as part of an Apache Arrow distribution.

Leave the store running in the background, start the processor:

```
$ python scripts/stream_processor.py /tmp/plasma
StreamProcessor waiting for calls at prefix 00000000
0.0 GB/s
0.0 GB/s
0.0 GB/s
...
```

It should show no traffic at the beginning (0.0 GB/s). This can be fixed by adding a streamer process:

```
$ python scripts/streamer.py /tmp/plasma
Store using prefix 00000001 for objects
Found processor StreamProcessor at prefix 00000000
Paylod size 80.0 MB
```

At which point you should be able to see data coming out the other end.

2.2 Nifty gridder

A highly accurate gridder implementation using an analytical kernel and 3D (de)gridding (see https://gitlab.mpcdf.mpg.de/ift/nifty_gridder). We utilise it here to check that we can correctly integrate a non-trivial processing component.

First start an Apache Plasma store and the processor as above:

```
$ plasma_store -s /tmp/plasma -m 1000000000 &
$ python scripts/ng_processor.py /tmp/plasma &
NiftyProcessor waiting for calls at prefix 00000000
```

Now we can run the `demo_wstack.py` test from the original repository, using the processor:

```
$ python scripts/demo_wstack.py /tmp/plasma
Store using prefix 00000001 for objects
Found processor NiftyProcessor at prefix 00000000
[...]
L2 error between explicit transform and gridder: 2.5686169932859304e-13
[...]
Testing adjointness of the gridding/degridding operation
adjointness test: 1.2946522165420342e-15
[...]
Testing adjointness of the gridding/degridding operation
adjointness test: 1.3392328015604763e-13
```

Demonstrating that all data gets transferred correctly.

CHAPTER

THREE

USAGE GUIDE

In the SKA SDP architecture, processing is meant to be orchestrated by execution engines, which perform I/O and delegate any actual work to processing functions. The data access library is therefore geared towards this type of working, with:

- **Processors** (see *processor*) passively providing processing functions
- **Callers** (see *caller*) actively managing the Plasma store and issuing calls to processors.

Processing functions are called by name, which are mapped to a parameter schema (see *make_call_schema()*).

3.1 A simple processor

A simple call schemas could look like follows:

```
import ska_sdp_dal as dal
import numpy as np
import pyarrow

TEST_SCHEMAS = [
    dal.make_call_schema('simple_fn', [
        dal.make_par('x', pyarrow.int64()),
        dal.make_tensor_input_par('ys', pyarrow.int64(), ['i']),
        dal.make_tensor_output_par('zs', pyarrow.int64(), ['i']),
    ])
]
```

This declares a processing function that takes an integer value for parameter `'x'`, a one-dimensional integer tensor for parameter `'ys'`, and writes a one-dimensional integer for parameter `'zs'`. We could define this processor as follows by sub-classing *Processor*:

```
class TestProcessor(dal.Processor):
    def __init__(self, **kwargs):
        super(TestProcessor, self).__init__(TEST_SCHEMAS, **kwargs)
    def simple_fn(self, x :int, ys :dal.TensorRef, zs :dal.TensorRef):
        zs.put(x + ys.get())
```

The *get()* method returns a `numpy.ndarray`, so the above method simply adds `x` to all values in `ys` and writes the result to `zs` using *put()*.

To allow running the procesor, we could define a simple event loop:

```
import sys

if __name__ == "__main__":
    proc = TestProcessor(plasma_path=sys.argv[1])
    while True:
        try:
            proc.process()
        except KeyboardInterrupt:
            exit(0)
```

Assuming we have started the processor with a backing Plasma store, we can now issue calls using a *Caller*:

```
store = dal.Store(plasma_path=sys.argv[1])
caller = dal.Caller(TEST_SCHEMAS, store)

result = caller.simple_fn(1, np.arange(100))
print(result['zs'].get())
```

Parameters can be passed both by name and by position here - in the latter case parameters are expected in the order they appear in the schema. If output parameters (here `zs`) are omitted, suitable references are automatically created and passed to the call. In either case, a dictionary with all output parameters is returned.

3.2 References

Both parameters passed to the processor as well as values returned by the caller are references to objects stored in Plasma. These objects are only copied into the shared memory space once, from there on out we can pass them around and use them as `numpy` arrays without incurring another copy.

This conversation happens automatically, but we can also manually allocate using the *Store* object:

```
ys = store.put_new_tensor(np.arange(100))
print(caller.simple_fn(1, ys)['zs'].get())
print(caller.simple_fn(2, ys)['zs'].get())
print(caller.simple_fn(3, ys)['zs'].get())
```

In this case, `ys` is only allocated in Plasma once, and passed to all calls without making a copy.

As the same principle applies to returned tensors, we can also do the following:

```
zs1 = caller.simple_fn(1, np.arange(100))['zs']
zs2 = caller.simple_fn(2, zs1)['zs']
zs3 = caller.simple_fn(3, zs2)['zs']
print(zs3.get())
```

In this case the tensor is passed from one function to the other purely using the Plasma store - the caller never touches it. Note that the caller especially will only wait once *get()* is called – it effectively works like a future in this context. Or put another way: We are effectively writing a small graph of processing function calls in the Plasma store. This is quite desirable to reduce the overhead of individual calls, however reduces control over the amount of memory used.

3.3 Batch Calls

To further reduce calling overhead, we can also issue many calls to the same processing function at the same time. The first example from the last section could also be written as follows:

```
ys = store.put_new_tensor(np.arange(100))
results = caller.simple_fn_batch([
  dict(x=1, ys=ys), dict(x=2, ys=ys), dict(x=3, ys=ys)
])
for result in results:
   print(result['zs'].get())
```

This will submit all three requests at the same time to the processor, which is slightly more efficient.

3.4 Broadcast

A *Caller* can issue calls to many processors at the same time:

```
import sys
import time

store = dal.Store(plasma_path=sys.argv[1])
caller = dal.Caller(TEST_SCHEMAS, store, broadcast=True, minimum_processors=0)

ys = store.put_new_tensor(np.arange(100))
for i in range(1000):
    caller.find_processors()
    results = caller.simple_fn(i, ys)
    print(f"Have {len(results} processors")
    time.sleep(1)
```

This caller will refresh its list of processors every second, selecting all the ones that accept the calls specified in `TEST_SCHEMAS`. The `broadcast` flag passed to *Caller* means that `simple_fn` will now return a list with results per call target.

Note that this clearly makes little sense for the processor we have constructed here. However, it might make sense in cases where we want to stream data to an unknown number of consumers without caring about the result. In this case, the recommended practice is to have zero-dimensional `tag` output parameters.

3.5 Scoping

Why do we need such a `tag` output parameter?

It is important to not forget that the caller is managing the lifecycle of all objects – if something goes out of scope at the caller, it will be removed from the Plasma store. Consider what would happen if we did not capture the result of a call:

```
caller.simple_fn(1, np.arange(100))
```

As this call happens asynchronously, the code will issue the call in the store and immediately return. As we are not holding onto any references to the call, the reference count on the returned reference will go to zero, which in turn means there's no reference to the issued call left. Therefore the issued call will be deleted immediately, possibly before a processor can pick it up.

This gives the caller a lot of control over what gets executed. For instance, we could do the following:

```
ys = store.put_new_tensor(np.arange(100))
results = [ caller.simple_fn(i, ys) for i in range(100) ]

time.sleep(0.01)
results = [ result['zs'].get(timeout=0) for result in results]
```

This will run for 10 ms, then collect any results that have been finished so far. Additionally, by overwriting the only remaining references to the remaining calls, any outstanding calls will be implicitly cancelled. Note that this would not work with batch calls, as the processor would pick up all invocations at the same time.

CHAPTER

FOUR

API DOCUMENTATION

The data access library is meant to enable the Science Data Processor to exchange tensor and table data between processing functions at high data rates. This is implemented using Apache Arrow data exchanged over a Apache Plasma shared memory object store.

4.1 Processing Functions

4.1.1 ska_sdp_dal.processor

Provides facilities for registering processing components

Each *Processor* registers a number of processing functions in the Plasma store, see *common.make_call_schema()*. You can especially use Plasma tensor and table objects as input and output parameters, see *common.make_tensor_input_par()*, *common.make_tensor_output_par()*, *common.make_table_input_par()* and common.make_table_output_par().

These processing functions can then be called from other processes using a *caller.Caller* instance connected to the same Plasma store.

class ska_sdp_dal.processor.**Processor**(*procs: List[pyarrow.Schema]*, *plasma_path: str*, *prefix: bytes = b''*, *name: str = None*)

A high-level processor interface.

Should be subclassed for implementing a concrete processor, with methods for every processing function registered by the processor. When *BasicProcessor.process()* is called, the processor will wait for a matching function call to appear in the Plasma store and input parameters to become available (see *BasicProcessor*).

All parameters are then automatically retrieved and unpacked into Python objects. References to the Plasma store are represented using *connection.TensorRef* or *connection.TableRef* instances. These can be used for getting and setting input and output tensors and tables respectively.

Parameters

- **procs** – Processing functions accepted by this processor
- **plasma_path** – Socket path of the Apache Plasma store
- **prefix** – Prefix to use for namespace.
- **name** – Name to use for registering the processor in the store. Defaults to the class name.

class ska_sdp_dal.processor.**LogProcessor**(*procs: List[pyarrow.Schema]*, *plasma_path: str*, *prefix: bytes = b''*, *name: str = None*)

Simple processor that just logs all calls.

Parameters

- **procs** – Processing functions accepted by this processor
- **plasma_path** – Socket path of the Apache Plasma store
- **prefix** – Prefix to use for namespace
- **name** – Name to use for registering the processor in the store. Defaults to the class name.

class ska_sdp_dal.processor.**BasicProcessor**(*procs: List[pyarrow.Schema]*, *plasma_path: str*, *prefix: bytes = b''*, *name: str = None*)

Low-level processor interface.

Deprecated: Use *Processor* instead.

Should be sub-classed for implementing a concrete processor. When *process()* is called, the processor will wait for a matching function call to appear in the Plasma store and input parameters to become available. Once that's the case, *_process_call()* will be called with the function name and a pyarrow.RecordBatch containing the parameters to the call.

Simple parameters can then be retrieved using :py:meth`parameter`, and input tensors using *tensor_parameter()*/*tensor_parameters()*. Output parameters can be set using *output_tensor()*

Parameters

- **procs** – Processing functions accepted by this processor
- **plasma_path** – Socket path of the Apache Plasma store
- **prefix** – Prefix to use for namespace
- **name** – Name to use for registering the processor in the store. Defaults to the class name.

abstract _process_call(*proc_func: str*, *batch: pyarrow.RecordBatch*)

oid_parameter(*batch: pyarrow.RecordBatch*, *name: str*, *allow_null: bool = False*) → pyarrow.plasma.ObjectID

Extract Object ID parameter from first row of record batch.

Parameters

- **batch** – Record batch containing parameter
- **name** – Name of parameter to extract
- **allow_null** – Value allowed to be null - will return None

output_tensor(*batch*, *name*, *array*, *typ=None*)

Write output tensor to storage

Note that this is less efficient than constructing it in-place, which we should support at some point (TODO)

Parameters

- **batch** – Record batch containing parameters
- **name** – Name of parameter to extract
- **arr** – Tensor as numpy array
- **typ** – Tensor value type

parameter(*batch: pyarrow.RecordBatch*, *name: str*, *typ: pyarrow.DataType = None*, *allow_null: bool = False*) → Any

Extract parameter from first row of record batch

Parameters

- **batch** – Record batch containing parameter
- **name** – Name of parameter to extract
- **typ** – Type to check (optional)
- **allow_null** – Value allowed to be null - will return None

process(*timeout=None*, *catch_exceptions=True*)

Attempts to process a call.

Blocks if no call is currently available.

Parameters

- **timeout** – Maximum time to block, in seconds
- **catch_exceptions** – Whether exceptions thrown by *_process_call()* should be caught and logged (the default).

Returns `False` if timeout expired, otherwise `True`

tensor_parameter(*batch: pyarrow.RecordBatch*, *name: str*, *typ: pyarrow.DataType = None*, *dim_names: List[str] = None*, *allow_null: bool = False*) → pyarrow.Tensor

Read tensor referred to via object ID parameter.

Parameters

- **batch** – Record batch containing parameters
- **name** – Name of parameter to extract
- **typ** – Tensor value type to check (optional)
- **dim_names** – Tensor dimensionality to check (optional)

tensor_parameters(*batch: pyarrow.RecordBatch*, *tensor_specs: List[Tuple[str, pyarrow.DataType, List[str], bool]]*) → List[pyarrow.Tensor]

Read tensors referred to via object ID parameters.

Parameters

- **batch** – Record batch containing parameters
- **tensor_specs** – Either list of strings or list of tuples of form (name, type, dimensionality, allow_null). If given, type and dimensionality will be checked. If allow_null is set, the object ID is allowed to be null, in which case None will get returned instead of a tensor.

4.1.2 ska_sdp_dal.caller

class ska_sdp_dal.caller.**Caller**(*procs: List[pyarrow.Schema]*, *store: ska_sdp_dal.store.Store*, *broadcast: bool = False*, *minimum_processors: int = 1*, *processor_prefix: bytes = b''*, *max_attempts: int = 100*, *verbose: bool = False*)

Base class for calls to a *processor.Processor* class

The constructor will create methods according to the passed call schemas - both for single and for batch calls. The batch variant will expect a list of dictionaries, see *batch_call()*.

Parameters

- **procs** – Call schemas to support. Will be used to find a compatible processor.
- **store** – Store area to use for calls (will use its Plasma client)

- **broadcast** – Send calls to all matching processors?
- **minimum_processors** – Raise an error if fewer processors are available
- **processor_prefix** – Allow changing processors after initialisation?
- **max_attempts** – Maximum attempts at resolving ObjectID collisions
- **verbose** – Log information about found processors

batch_call(*call_schema: pyarrow.Schema*, *calls: List[Dict[str, Any]]*) → List[Dict[str, *ska_sdp_dal.connection.TensorRef*]]

Create a number of calls to a function with the given schema.

Parameters

- **call_schema** – Schema of the call
- **calls** – List of parameter dictionaries

Returns List of output parameter dictionaries per call (if broadcasting also per processor)

call(*call_schema: pyarrow.Schema*, **args*, ***kwargs*) → pyarrow.plasma.ObjectID

Create a number of calls to a function with the given schema.

Both positioned and keyword arrays are supported, using the position and name of the parameter in the schema, respectively.

Parameters

- **call_schema** – Schema of the call
- **args** – List of parameters
- **kwargs** – Dictionary of parameters

find_processors(*verbose=False*)

Locate compatible processors.

Done automatically when the caller is constructed. Call again to refresh the list of processors to call. Typically used with broadcasting callers.

Parameters **verbose** – Log information about found processors

property num_processors

The number of processors located by this caller

4.2 Storage

4.2.1 ska_sdp_dal.store

class ska_sdp_dal.store.**Store**(*plasma_path: str*, *max_attempts: int = 10000*, *name: str = None*)

A storage namespace within a Plasma store

Used for holding shared data objects, such as tensors and tables. These can be passed to processors.

property conn

make_tensor_ref(*oid: pyarrow.plasma.ObjectID*, *typ: pyarrow.DataType = None*, *dim_names: List[str] = None*) → *ska_sdp_dal.connection.TensorRef*

Create a TensorRef object for an existing object in Plasma

Parameters

- **oid** – Existing object ID
- **typ** – Element datatype. If *ComplexType*, will convert.
- **dim_names** – Dimension names

Returns Reference to tensor

new_table_ref(*schema: pyarrow.Schema = None*) → *ska_sdp_dal.connection.TableRef*

Allocate an Object ID for a new tensor in Plasma

Parameters

- **typ** – Element datatype. If *ComplexType*, will convert.
- **dim_names** – Dimension names

Returns Reference to tensor

new_tensor_ref(*typ: pyarrow.DataType = None*, *dim_names: List[str] = None*) → *ska_sdp_dal.connection.TensorRef*

Allocate an Object ID for a new tensor in Plasma

Parameters

- **typ** – Element datatype. If *ComplexType*, will convert.
- **dim_names** – Dimension names

Returns Reference to tensor

put_new_table(*table: Union[pyarrow.Table, pandas.DataFrame, Mapping[str, pyarrow.ChunkedArray], Mapping[str, pyarrow.Array], Mapping[str, list]]*, *schema: pyarrow.Schema = None*) → *ska_sdp_dal.connection.TableRef*

Allocate and create a new table in Plasma

See `connection.TableRef.put()` for notes about possible parameters.

Parameters

- **table** – Table data
- **schema** – Table schema

Returns Reference to table

put_new_tensor(*arr: numpy.ndarray*, *typ: pyarrow.DataType = None*, *dim_names: List[str] = None*) → *ska_sdp_dal.connection.TensorRef*

Allocate and create a new tensor in Plasma

Parameters

- **arr** – Data as numpy array
- **typ** – Element datatype. If ComplexType, will convert.
- **dim_names** – Dimension names

Returns Reference to tensor

4.2.2 ska_sdp_dal.connection

class ska_sdp_dal.connection.**Connection**(*plasma_path: str*)
A connection to a Plasma store.

Subscribes to events and uses it to maintain a list of objects in the store. We especially track namespaces.

property client

get_buffers(*oids*, *timeout=None*) → pyarrow.plasma.PlasmaBuffer
Retrieve object for given OIDs.

Uses a cache to prevent duplicated requests to the Plasma store.

Parameters

- **oids** – Plasma object IDs
- **timeout** – Time to wait for buffers to become available

Returns Plasma buffer

get_ref_buffers(*refs: List[Ref]*, *timeout: float = None*, *auto_delete: bool = True*) → None
Retrieves the buffers for multiple Plasma references at a time.

Blocks as long as any (!) of the objects have not been created yet.

Parameters

- **refs** – References to retrieve buffer of
- **timeout** – Maximum time this function is allowed to block.
- **auto_delete** – Delete object in store when reference is dropped

Raises TimeoutException

property namespace_meta

property namespace_procs

property namespaces

object_exists(*oid*) → bool
Checks whether the given object ID is known to exist

Parameters **oid** – Object ID to check

object_size(*oid*) → bool
Gets the size of the given object

Parameters **oid** – Object ID to check

reserve_namespace(*name: str = None*, *procs: List[pyarrow.Schema] = []*, *prefix: bytes = b''*) → Tuple[bytes, pyarrow.plasma.PlasmaBuffer]
Reserve a new namespace within the Plasma store

This will automatically clear all objects with the given prefix

Parameters

- **name** – Informative display name for namespace
- **procs** – Call schemas supported (if any)
- **name** – Metadata to associate with schema
- **prefix** – Prefix for prefix

Returns Prefix, buffer with declaration (to keep namespace alive)

update_obj_table(*timeout: float = 0*)
Update the object table

Parameters **timeout** – If given, allow blocking for up to the given time or until the next update happens.

Returns A list of received update notifications

class ska_sdp_dal.connection.**Ref**(*conn: ska_sdp_dal.connection.Connection, oid: pyarrow.plasma.ObjectID, auto_delete: bool = True, dependencies: List[Ref] = [], references: List[Ref] = []*)
Refers to an object in storage

Subclassed by type. Might not have been created yet. Can have two kinds of relationships with other objects:

- dependency: Object is required for this object to be created. Must ensure objects stay alive *until* this object is found to be created. Typically refers to call objects.
- reference: Object that is referenced from this object and must therefore be kept alive while this object is still needed.

add_dependency(*ref: ska_sdp_dal.connection.Ref, timeout: float = 0*) → None
Registers the identified object as a dependency.

This will ensure that the object is kept alive *until* we have retrieved the data for this object. Blocks if the object does not yet exist in the store.

Parameters

- **ref** – Reference to register as dependency
- **timeout** – Maximum time this method is allowed to block.

Raises TimeoutException

add_reference(*ref: Union[Ref, pyarrow.plasma.ObjectID], timeout: float = 0*) → None
Registers the identified object as referenced

This will ensure that the object is kept alive as long as this object is referenced. Might Block if the object does not yet exist in the Plasma store.

Parameters

- **ref** – Reference to register as referenced
- **timeout** – Maximum time this method is allowed to block.

Raises TimeoutException

get_buffer(*timeout: float = None, auto_delete: bool = True*) → pyarrow.Buffer
Get Arrow buffer for this Plasma reference.

Blocks if the object has not yet been created. :param timeout: Maximum time this method is allowed to block. :param auto_delete: Delete object in store when reference is dropped :raises: TimeoutException

property oid

class ska_sdp_dal.connection.**TableRef**(*conn: ska_sdp_dal.connection.Connection, oid: pyarrow.plasma.ObjectID, schema: pyarrow.Schema = None, auto_delete: bool = True*)
Refers to a Table in Plasma store.

Might not have been created yet - wraps an Object ID and expected type information.

get(*timeout=None*)
Get Arrow table

Parameters **timeout** – How long the function is allowed to block if object does not exist yet

get_awkward()
Get table as Awkward array

This can generally be done without copying the data.

get_dict()
Get table as Python dictionary

get_pandas(**args*, ***kwargs*)
Get table as Pandas dataframe

Parameters **kwargs** – Parameters to panda conversion. See `pyarrow.Table.to_pandas()`.

put(*table: Union[pyarrow.Table, pandas.DataFrame, Mapping[str, pyarrow.ChunkedArray], Mapping[str, pyarrow.Array], Mapping[str, list]]*, *max_chunksize=None*)
Write the given table into storage.

The table can be given as pandas `DataFrame`, dictionary of strings to `pyarrow.Array` or lists, which will be converted into the equivalent table. If the arrays are chunked, the chunks of all columns must match. See also `pyarrow.table()`.

Parameters

- **table** – Table to write.
- **max_chunksize** – Maximum size of record batches to split table into

property schema

class ska_sdp_dal.connection.**TensorRef**(*conn: ska_sdp_dal.connection.Connection*, *oid: pyarrow.plasma.ObjectID*, *typ: pyarrow.DataType = None*, *dim_names: List[str] = None*, *auto_delete: bool = True*)
Refers to a tensor in object storage.

Might not have been created yet - wraps an Object ID and expected type information.

property dim_names

get(*timeout=None*)
Retrieve the tensor from storage. Might block.

Parameters **timeout** – Maximum time this method is allowed to block.

put(*arr: numpy.ndarray = None*)
Write the given value into storage.

Parameters **arr** – Array to write to storage. Empty by default.

property typ

exception ska_sdp_dal.connection.**TimeoutException**(*refs*, *timeout*)

property refs

property timeout

4.2.3 ska_sdp_dal.common

class ska_sdp_dal.common.**ComplexType**(*real_type*, *complex_dtype*, *real_dtype*)
Pseudo-type to refer to complex values.

Use in place of arrow types.

to_pandas_dtype()

ska_sdp_dal.common.**NAMESPACE_DECL_SUFFIX = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x**
First object in namespace is expected to be its declaration

ska_sdp_dal.common.**NAMESPACE_ID_SIZE = 4**
Bytes of the Object ID we are going to use as namespace prefix

ska_sdp_dal.common.**OBJECT_ID_SIZE = 20**
Length of Plasma's object IDs

ska_sdp_dal.common.**PROC_NAMESPACE_ARGV_META = b'proc:argv'**
Metadata entry for namespace process arguments

ska_sdp_dal.common.**PROC_NAMESPACE_META = b'proc:namespace'**
Metadata entry for namespace name

ska_sdp_dal.common.**PROC_NAMESPACE_PID_META = b'proc:pid'**
Metadata entry for namespace process ID

ska_sdp_dal.common.**call_name**(*schema: pyarrow.Schema*) → str
Get call name from call schema

Parameters **schema** – Call schema

ska_sdp_dal.common.**complex128 = <ska_sdp_dal.common.ComplexType object>**
Double-precision complex type

ska_sdp_dal.common.**complex64 = <ska_sdp_dal.common.ComplexType object>**
Single-precision complex type

ska_sdp_dal.common.**from_numpy_dtype**(*dtype: Any*) → pyarrow.DataType

ska_sdp_dal.common.**is_namespace_decl**(*oid: pyarrow.plasma.ObjectID*)
Checks whether the given object ID declares a namespace.

ska_sdp_dal.common.**make_call_schema**(*func_name: str, pars: List[NewType.<locals>.new_type], metadata={}*) → pyarrow.Schema
Create schema for calls through the Plasma store.

Parameters

- **func_name** – Function name
- **pars** – List of parameters
- **metadata** – Metadata to associate with schema

ska_sdp_dal.common.**make_oid_input_par**(*name: str, nullable: bool = False, metadata: Dict[bytes, bytes] = {}*) → NewType.<locals>.new_type
Create input Object ID parameter to pass to make_call_schema.

Marking the parameter as input means that the call will be delayed until an object with the given ID appears in the Plasma store.

Parameters

- **name** – Parameter name
- **nullable** – Allowed to be null?
- **metadata** – Metadata dictionary to associate with field.

ska_sdp_dal.common.**make_oid_output_par**(*name: str*, *nullable: bool = False*, *metadata: Dict[bytes, bytes] = {}*) → NewType.<locals>.new_type

Create Object ID parameter to pass to make_call_schema.

The call will be skipped if all outputs already exist in the Plasma store.

Parameters

- **name** – Parameter name
- **nullable** – Allowed to be null?
- **metadata** – Metadata dictionary to associate with field.

ska_sdp_dal.common.**make_oid_par**(*name: str*, *nullable: bool = False*, *metadata: Dict[bytes, bytes] = {}*) → NewType.<locals>.new_type

Create Object ID parameter to pass to make_call_schema.

Parameters

- **name** – Parameter name
- **nullable** – Allowed to be null?
- **metadata** – Metadata dictionary to associate with field.

ska_sdp_dal.common.**make_par**(*name: str*, *typ: pyarrow.DataType*, *nullable: bool = False*, *metadata: Dict[str, str] = {}*) → NewType.<locals>.new_type

Create parameter declaration to pass to make_call_schema.

Parameters

- **name** – Parameter name
- **typ** – Arrow data type
- **nullable** – Allowed to be null?
- **metadata** – Metadata dictionary to associate with field.

ska_sdp_dal.common.**make_table_input_par**(*name: str*, *table_schema: pyarrow.Schema*, *nullable: bool = False*) → NewType.<locals>.new_type

Create input tensor parameter to pass to make_call_schema.

Marking the parameter as input means that the call will be delayed until a tensor with the given ID appears in the Plasma store.

Parameters

- **name** – Parameter name
- **elem_type** – Tensor element type
- **dim_names** – Dimension names
- **nullable** – Allowed to be null?

ska_sdp_dal.common.**make_tensor_input_par**(*name: str*, *elem_type: pyarrow.DataType*, *dim_names: List[str]*, *nullable: bool = False*) → NewType.<locals>.new_type

Create input tensor parameter to pass to make_call_schema.

Marking the parameter as input means that the call will be delayed until a tensor with the given ID appears in the Plasma store.

Parameters

- **name** – Parameter name
- **elem_type** – Tensor element type
- **dim_names** – Dimension names
- **nullable** – Allowed to be null?

ska_sdp_dal.common.**make_tensor_output_par**(*name: str*, *elem_type: pyarrow.DataType*, *dim_names: List[str]*, *nullable: bool = False*) → NewType.<locals>.new_type

Create input tensor parameter to pass to make_call_schema.

Marking the parameter as input means that the call will be delayed until a tensor with the given ID appears in the Plasma store.

Parameters

- **name** – Parameter name
- **elem_type** – Tensor element type
- **dim_names** – Dimension names
- **nullable** – Allowed to be null?

ska_sdp_dal.common.**object_id_hex**(*oid: pyarrow.plasma.ObjectID*) → str

Convert Object ID into a hexadecimal string representation

Parameters **oid** – The Object ID to convert (as bytearray or ObjectID)

ska_sdp_dal.common.**objectid_generator**(*prefix: bytes*, *size: int = 20*) → Iterator[bytes]

Generate ObjectIDs with a given prefix.

Parameters **prefix** – Prefix as binary string

ska_sdp_dal.common.**par_meta**(*field: pyarrow.Field*) → Optional[str]

Get parameter kind metadata from schema field

Parameters **schema** – Field

Returns Parameter kind, or *None* if not set

ska_sdp_dal.common.**par_table_schema**(*field: pyarrow.Field*) → Optional[pyarrow.Schema]

Get table schma for a parameter

Parameters **field** – Field to read metadata frmo

Returns Table schema, or *None* if not set

ska_sdp_dal.common.**par_tensor_dim_names**(*field: pyarrow.Field*) → Optional[List[str]]

Get tensor element type parameter

Parameters **field** – Field to read metadata frmo

Returns Parameter element type, or *None* if not set

ska_sdp_dal.common.**par_tensor_elem_type**(*field: pyarrow.Field*) → Optional[pyarrow.DataType]

Get tensor element type parameter

Parameters **field** – Field to read metadata from

Returns Parameter element type, or *None* if not set

ska_sdp_dal.common.**parse_hex_objectid**(*oid_str: str*) → bytes

Parse an Object ID given as a hexadecimal string representation

Note that this allows Object IDs to have less than 20 bytes, i.e. partial Object IDs (prefixes) are parsed without error.

Parameters **oid_str** – String representation

Returns Object ID as binary string

ska_sdp_dal.common.**schema_compatible**(*expected: pyarrow.Schema*, *actual: pyarrow.Schema*) → bool

Checks for compatibility between (call) schemas.

This means that all expected fields are there and have the same types (including relevant metadata).

Parameters

- **expected** – Expected schema
- **actual** – Schema to check

Returns Empty list if compatible, otherwise list of mismatches

CHAPTER

FIVE

PROJECT-NAME DOCUMENTATION HEADING

These are all the packages, functions and scripts that form part of the project.

- *API Documentation*

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

F

G

I

L

M

N

O

P

R

S

T

U