ska-sdp-plasmastman

Release 1.3

Rodrigo Tobar

Oct 06, 2022

CONTENTS

CHAPTER

ONE

INSTALLATION

These instructions are sufficient to build and install `PlasmaStMan`, making it readily available for usage by third-party applications.

While these instructions are enough to get started, attention needs to be paid when planning to use this storage manager from a python environment. For details see *Python quirks*.

1.1 Dependencies

This project depends on:

- Any C++14 compiler
- casacore > 3.3.0 (with 64-bit table support)
- arrow >= 1.0.0-SNAPSHOT with plasma support

1.2 Compiling

This is a `cmake`-based project, so it can be built as any standard `cmake` project:

```
$ git clone https://gitlab.com/ska-telescope/ska-sdp-plasmastman
$ cd plasma-storage-manager
$ cmake . -B build
# cmake --build build
```

Some of the most relevant `cmake` variables (passed on the first `cmake` invocation via `-Dvariable=value`) used for compiling are:

- `CASACORE_ROOT_DIR`: Root of arbitrary casacore installations in case one is used
- `Arrow_DIR`: directory containing the `cmake` configuration exported by Apache Arrow (usually under `lib/cmake/arrow` in the arrow installation area).
- `Plasma_DIR`: directory containing the `cmake` configuration exported by Apache Plasma (usually under `lib/cmake/arrow` in the arrow installation area).
- `CMAKE_CXX_COMPILER`: The C++ compiler to use.
- `CMAKE_CXX_FLAGS`: Extra C++ compilation flags.
- `CMAKE_BUILD_TYPE`: The type of build to produce, one of `Debug`, `Release` and `RelWithDebInfo`.
- `BUILD_TESTING`: Whether to build unit tests or not, defaults to `ON`.

1.3 Testing

A set of unit tests is included and built by default. To execute them do:

```
$ cmake --build build --target test
```

The unit tests require the `plasma-store-server` executable (part of a standard C++ Arrow Plasma installation) to be visible in the path.

If you want further control on `ctest`'s command line flags you can do:

```
$ cmake --build build --target test -- ARGS="<ctest command line flags>"
```

or alternatively:

```
$ cd build/
$ ctest <ctest command line flags>
```

1.4 Python quirks

When using `PlasmaStMan` from python, special attention needs to be paid to how the `python-casacore` and `pyarrow` python packages, if needed by your python code, are installed to avoid some otherwise difficult to debug errors.

1.4.1 python-casacore

TL;DR:

- Don't install the pre-built binary wheels from PyPI.
- If you can, use the kernsuite repositories to install the `casacore` libraries and `python-casacore` python package from pre-built apt packages.
- If installing from kernsuite is not an option, then ensure `python-casacore` is built against the same `casacore` installation `PlasmaStMan` was built against.

Starting from version 3.4.0, the `python-casacore` package offers pre-built binary wheels for some major OS and python version combinations. These binary wheels come bundled with a copy of the underlying casacore libraries (`libcasa_casa.so`, `libcasa_tables.so`, etc) and their dependencies. Each of these bundled libraries actually have a specific `SONAME` s and matching filesname (e.g. `libcasa_tables-734048a7.so.6`), thus avoiding interfering with any system-wide installation.

On the other hand, the plug-in mechanism used to register third-party storage managers with `casacore` involves first loading the storage manager shared library into memory, then invoking a registration function in the library that registers itself into a static `casacore`-owned registration map, and finally checking that the registration was successful. This usually looks like this:

```
+-------------+      1. dlopen()     +-----------------+
| casacore.so | -------------------> | plasmastman.so |
+-------------+                      +-----------------+
 ^  |  ^  |                              ^  |
 |  |  |  |   2. register_plasmastman()   |  |
 |  |  |  \-------------------------------/  |
 |  |  |                                     |
```

(continues on next page)

(continued from previous page)

```
 |  |  |     3. DataMan::registerCtor()      |
 |  |  \-------------------------------------/
 |  |
 \--/  4. check_registration() // all good :)
```

However when using the binary wheels from PyPI, and because of the difference in `SONAME` between the bundled libraries and the libraries used to compile the storage manager, two different copies of `casacore.so` are loaded into memory, and the interaction looks like this:

```
+----------------------+   1. dlopen()     +----------------+  1.1 dlopen()  +-----------
↪--+
| casacore-734048a7.so | ----------------> | plasmastman.so | -------------> | casacore.
↪so |
+----------------------+                   +----------------+                +-----------
↪--+
          ^  |     |                            ^  |                                ^
          |  |     | 2. register_plasmastman() |  |                                |
          |  |     \---------------------------/  |                                |
          |  |                                    |                                |
          |  |                                    |    3. DataMan::registerCtor()  |
          |  |                                    \--------------------------------/
          |  |
          \--/  4. check_registration() // fails, registration cannot be found :(
```

In particular, the error message will look something like:

```
RuntimeError: Table DataManager error: Data Manager class PlasmaData is not registered
```

This situation is specific to the binary wheels distributed via PyPI. To avoid this issue one must ensure that the `python-casacore` package uses the same libraries the storage manager was compiled against. This could be done either by installing `python-casacore` from source and pointing it to an existing `casacore` installation (which itself might be installed from source or not), or by using pre-compiled packages that don't incur into this duplication of libraries, like the apt packages provided by the Kernsuite project.

1.4.2 pyarrow

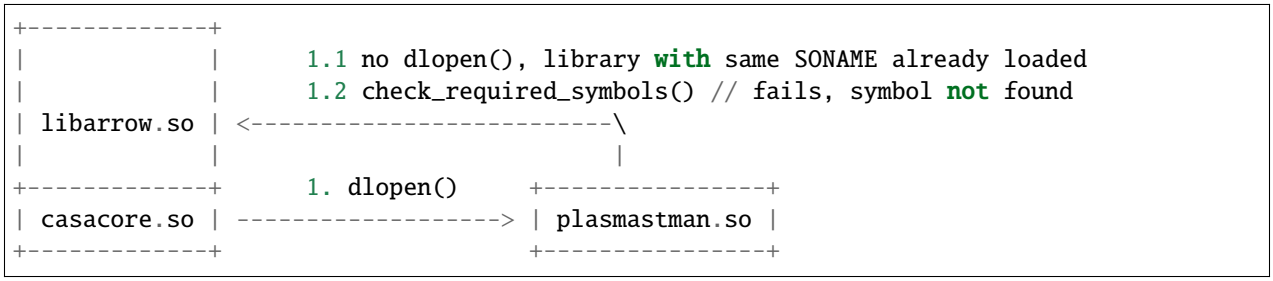
TL;DR:

- Pre-built binary wheels from PyPI are incompatible with pre-built Arrow apt packages provided by Apache.
- You can install a *different* version of `pyarrow` alongside the pre-built Arrow apt packages, but this might break in the future.
- You can install `pyarrow` from sources, building them against the same Arrow/Plasma installation `PlasmaStMan` was built against.

Apache Arrow makes available binary wheels in PyPI for users to install the `pyarrow` python package without needing a compiler or any other external libraries. Like in the case of `python-casacore`, these binary wheels are bundled with their own copy of the Arrow shared libraries (`libarrow.so`, `libplasma.so` and so on). For a given version of Arrow, these libraries share the same `SONAME` with those installed via the Arrow apt repositories. However, the PyPI `pyarroww` binary wheels are compiled using a version of gcc prior to the introduction that didn't offer a dual ABI mechanism (read the link for a more detailed explanation). The effect this has is that the arrow libraries generated by newer versions of `gcc` define differently named symbols than those generated by older versions of `gcc`, and therefore they cannot be mixed freely (e.g., linked or dynamically loaded). This problem has been reported, but other than acknowledging

the issue and providing some suggestions on how to proceed, the final response was that this use case is not officially supported by the Arrow published artifacts.

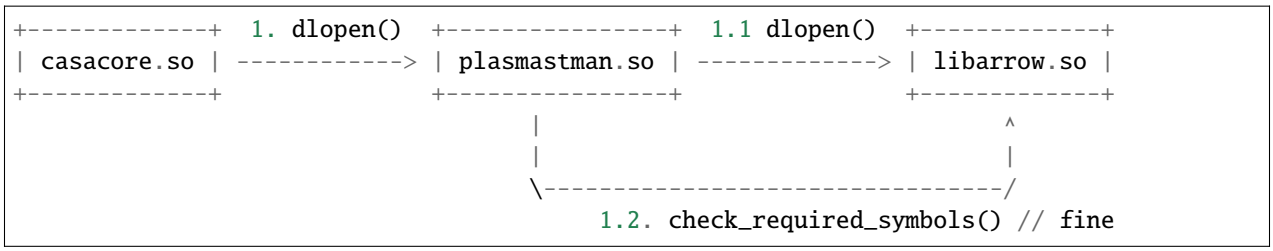
Because of this situation, problems occur if the `python` process loads the storage manager, which has been compiled against the apt-installed Arrow libraries, *after* importing the PyPI-installed `pyarrow`. In such cases the following situation occurs:



In particular, the error message will look something like:

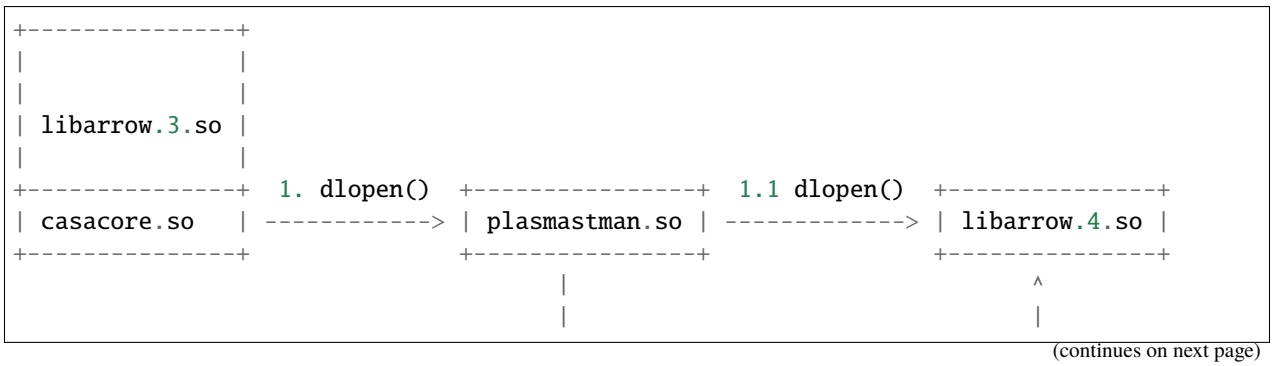
```
RuntimeError: Shared library plasmastman not found in CASACORE_LDPATH or (DY)LD_LIBRARY_
↪PATH
libcasa_plasmastman.so.4: cannot open shared object file: No such file or directory
libcasa_plasmastman.so: cannot open shared object file: No such file or directory
libplasmastman.so.4: cannot open shared object file: No such file or directory
/usr/local/lib/libplasmastman.so: undefined symbol: _ZN5arrow5fieldENSt7__cxx1112basic_
↪stringIcSt11char_traitsIcESaIcEEESt10shared_ptrINS_8DataTypeEEbS6_IKNS_
↪16KeyValueMetadataEE
```

Note that if `pyarrow` has not yet been imported at the time the storage manager library is loaded then no error occurs:



The situation above is a bit brittle as it depends on `pyarrow` not being loaded at the time. Moreover, loading it later might also lead to the same missing symbol error.

A possibility, somewhat fragile, is to install a version of `pyarrow` from PyPI *different* to that installed via apt so the `SONAME` of both libraries don't collide. That way, `plasmastman.so` is forced into loading a different copy of the arrow library into memory. This results in the following:



(continues on next page)

(continued from previous page)

```
                                    \-----------------------------------/
                                         2. check_required_symbols() // all good :)
```

This obviously results in two copies of different versions of the Arrow library loaded into memory. Although we haven't noticed any side-effects, this might not always be the case.

The ultimate solution is of course to avoid the problem with bundled libraries altogether and install `pyarrow` from source, compiling against the same installation of Arrow/Plasma the `PlasmaStMan` was compiled against. This results on a clean environment, but has a higher setup cost:

```
+-------------+
|             |      1.1 no dlopen(), library with same SONAME already loaded
|             |      1.2 check_required_symbols() // all good :)
| libarrow.so | <--------------------------\
|             |                            |
+-------------+      1. dlopen()     +-----------------+
| casacore.so | -------------------> | plasmastman.so |
+-------------+                      +-----------------+
```

CHAPTER

TWO

USAGE

`PlasmaStMan` maps Apache Arrow Tensors and Tables (i.e., their Object IDs in the Plasma store) to individual columns within a casacore Table.

Arrow Tensors map directly to casacore Columns one to one. The mapping then consists on a pair of strings indicating the Object ID of the Tensor in the Plasma store and the name of the casacore Table column it provides data to. Checks are in place to ensure that a Tensor's shape and type match those of the corresponding column of the casacore Table. All casacore data types are supported by this mapping with the exception of `Strings`.

Arrow Tables on the other hand contain one or more Fields, which individually map to casacore Columns. The mapping then consists on a pair of strings indicating the ObjectID of the Table in the Plasma store and the name of the Field that should be considered, which should match the name of the casacore Table column it provides data to. Like in the case of Tensors, a Field's shape (length) and type are checked against those of the corresponding column of the casacore Table. Columns in an Arrow Table have only a single dimension, so they are currently only supported as scalar columns. Additionally, Complex values are not supported natively by Arrow Tables, and therefore `Complex` and `DComplex` values are supported as Arrow `Struct` objects with `r` and `i` fields.

2.1 Configuration

`PlasmaStMan` always needs to connect to a Plasma store. This happens through a Unix socket in the filesystem. The location of this socket defaults to `/tmp/plasma`, but its value can be overriden by setting the `PLASMA_SOCKET` environment variable.

Either when reading or writing, certain aspects of `PlasmaStMan` can be configured at runtime via Storage manager *properties* (arbitrary key-value pairs). `PlasmaStMan` supports the following properties:

- `PLASMACONNECTRETRIES`: the number of times the Plasma client should try to connect to the Plasma store before giving up. Defaults to `50`.
- `PLASMAGETTIMEOUT`: the timeout in milliseconds to use when getting an object from the Plasma store that is not immediately available. Defaults to `10000`.

2.2 Reading

When reading data from a Table backed by a `PlasmaStMan` storage manager users need to ensured that the `libplasmastman` shared library is visible in the dynamic linker's path (e.g., adding the directory containing the library to the `LD_LIBRARY_PATH` environment variable in Linux).

Other than this, existing casacore-based applications do not require any modification or recompilation.

2.3 Writing

Note: At the moment `PlasmaStMan` **does not support** writing data to plasma.

Writing is a trickier business.

Even though the data itself cannot be written *through* `PlasmaStMan`, what can currently be done is creating a casacore table that points to existing data in Plasma. To achieve this one must inform the storage manager about the mapping between Object IDs and columns. This can be done in two different ways:

- If writing a program in C++, one can use the *PlasmaStMan* class to create the storage manager object and bind it to tables. The main constructor of this class accepts two `std::map` objects to provide the mapping from Object ID to column name for Tensors and Tables.
- Storage managers allow *specifications* to be given at creation time. This includes the *properties* specified above, along with the following additional keys:
  - `PLASMASOCKET`: the Unix socket used to connect to Plasma, override the `PLASMA_SOCKET` environment variable.
  - `TENSOROBJECTIDS`: a casacore `Record` object (i.e., a mapping) where keys are Tensor Object IDs and values are column names.
  - `TABLEOBJECTIDS`: a casacore `Record` object (i.e., a mapping) where keys are Table Object IDs and values are column names.

  Because this is a generic mechanism, these specifications can be given through different interfaces. For example, the `TaQL` language supports the creation of tables with a given Data Manager specification (see section 8.2, *Data manager specification*). The `python-casacore` python bindings also allow the creation of tables with specific Data Manager inforation (see `dminfo` argument).

2.4 Example

Note: This example needs *pyarrow* installed.

Included in the `ska-sdp-plasmastman` repository is a python-based script that demonstrates how to create a casacore Table pointing to Plasma-stored Tensors and Tables. This can be used to test `PlasmaStMan` from external programs:

```
# Start a plasma store and store tensor and table data with arbitrary values
# and create a table pointing to this new data (using taql).
# Use -h to see a bit more of information on how to use it
$> python scripts/plasma_writer.py -o <table_name> -t <tensor1> -t <tensor2> -T <table1>␣
↪... &

# Make the new storage manager visible to third-party apps
$> export LD_LIBRARY_PATH=your-build-directory/src/ska/plasma

# Read the table metadata with casacore's showtableinfo
$> showtableinfo in=<table_name>

# Read the table data back with casacore's taql
$> taql 'select * FROM <table_name>'
```

CHAPTER

THREE

CHANGELOG

- Migrated to use new cpp-build-base image and new $SKA_CPP_DOCKER_BUILDER_IMAGE variable.

3.1 1.3

- Added support for runtime *properties* on the plasma storage manager. Two properties are supported, `PLASMACONNECTRETRIES` and `PLASMAGETTIMEOUT`, making it possible to configure plasma-related aspects of the storage manager at runtime.
- Added validation for user-provided Plasma Object IDs.

3.2 1.2

- Added support for Arrow Table mapping. Individual Fields/Columns from an Arrow Table can be mapped to the equally named casacore Table. The mapping can be given via the new `TABLEOBJECTIDS` Data Manager specification property.
- Changed `OBJECTIDS` Data Manager specification property name to `TENSOROBJECTIDS` to explicitly state what type of objects do they refer to.
- Added public C++ API documentation where missing.

3.3 1.1

- Added support for generic configuration of the plasma storage manager via Data Manager Specification (casacore `Record`) objects. This makes it possible to create casacore Tables with correctly configured plasma storage managers without executables built for that specific purpose. Most unit tests indeed now create Tables using `taql`, which supports this generic configuration mechanism.

3.4 1.0.1

- Removed memcheck tests from GitLab CI pipeline.

3.5 1.0

- First version of the plasma storage manager.
- A single column is backed up by a single Tensor stored on a single Plasma store; multiple columns require multiple Tensors stored on a single Plasma store.
- Read-only operations are supported for both scalar and array columns.
- Shape and type are checked to ensure a Tensor can be used for a given column.
- Zero-copy is supported for operations where this can be accomplished, namely: full-column reads (array and scalar columns), single, continuous row range reads (array and scalar columns), and single cell reads (array columns).
- Existing programs can use this storage manager without modifications, as demonstrated by tests with `taql` and `showtableinfo`.
- Table creation is a manual process. A `table_writer` utility is included to help with this.

CHAPTER

FOUR

API

4.1 Casacore classes

ska::plasma::PlasmaStMan and *ska::plasma::PlasmaStManColumn* are the two main classes implementing the Storage Manager API as mandated by casacore.

class **PlasmaStMan** : public DataManager

The Plasma-based storage manager

This is implemented using a pimpl idiom to hide the particulars of the implementation and hide it from users.

Public Functions

PlasmaStMan(std::string plasma_socket = "", const std::map<std::string, *ObjectID*> &tensor_object_ids = {}, const std::map<std::string, *ObjectID*> &table_object_ids = {})

Creates a new instance of the Plasma Storage Manager connected to the given socket, and mapping columns to Arrow Tensors and Tables as indicated in the given mappings.

Parameters

- **plasma_socket** – The UNIX socket where the Plasma store listens for connections. If not given, or empty, it defaults to `/tmp/plasma`, unless the `PLASMA_SOCKET` environment variable is set, in which case its value takes precedence.
- **tensor_object_ids** – A mapping from column names to Object IDs in the Plasma store where Arrow Tensors with the data for the respective column can be found.
- **table_object_ids** – A mapping from column names to Object IDs in the Plasma store where Arrow Tables with the data for the respective column can be found (the name of the column being mapped must be the same as the column name in the Arrow Table).

~PlasmaStMan()

Destructor declaration because of the pimpl idiom, otherwise its implementation is defaulted.

void **ping_plasma**()

See also:

PlasmaClient::ping

void **set_plasma_get_timeout**(std::int64_t timeout)

See also:

PlasmaClient::set_get_timeout

void **set_plasma_connect_retries**(int connect_retries)

See also:

PlasmaClient::set_connect_retries

Public Static Functions

static casacore::DataManager ***makeObject**(const casacore::String &aDataManType, const casacore::Record &spec)

Factory function invoked by casacore to create an instance of *PlasmaStMan* from a given DataManager specification.

See also:

PlasmaStMan::impl::dataManagerSpec

Parameters

- **aDataManType** – The name of the data manager.
- **spec** – The specification of the data manager.

Returns
A new *PlasmaStMan* object.

class **impl**

The Plasma-based storage manager implementation

This class fully implements the plasma-based storage manager, while *PlasmaStMan* only exposes this implementation, while hiding its dependencies.

Public Functions

impl(std::string plasma_socket = "", std::map<std::string, *ObjectID*> tensor_object_ids = {}, std::map<std::string, *ObjectID*> table_object_ids = {})

See also:

PlasmaStMan::PlasmaStMan

~impl()

Destructor declaration because of incomplete *PlasmaStManColumn* type usage in one of our members; otherwise its implementation is defaulted.

void **ping_plasma**()

See also:

PlasmaStMan::ping_plasma

void **set_plasma_get_timeout**(std::int64_t timeout)

See also:

PlasmaStMan::set_plasma_get_timeout

void **set_plasma_connect_retries**(int connect_retries)

See also:

PlasmaStMan::set_plasma_connect_retries

DataManager ***clone**() const

See also:

PlasmaStMan::clone

String **dataManagerType**() const

See also:

PlasmaStMan::dataManagerType

String **dataManagerName**() const

See also:

PlasmaStMan::dataManagerName

void **create64**(rownr_t aNrRows)

See also:

PlasmaStMan::create64

rownr_t **open64**(rownr_t aRowNr, AipsIO &ios)

See also:

PlasmaStMan::open64

rownr_t **resync64**(rownr_t aRowNr)

See also:

PlasmaStMan::resync64

Bool **flush**(AipsIO&, Bool doFsync)

See also:

PlasmaStMan::flush

DataManagerColumn ***makeScalarColumn**(const String &aName, int aDataType, const String &aDataTypeID)

See also:

PlasmaStMan::makeScalarColumn

DataManagerColumn ***makeDirArrColumn**(const String &aName, int aDataType, const String &aDataTypeID)

See also:

PlasmaStMan::makeDirArrColumn

DataManagerColumn *makeIndArrColumn(const String &aName, int aDataType, const String &aDataTypeID)

See also:

PlasmaStMan::makeIndArrColumn

void deleteManager()

See also:

PlasmaStMan::deleteManager

void addRow64(rownr_t aNrRows)

See also:

PlasmaStMan::addRow64

Record dataManagerSpec() const

See also:

PlasmaStMan::dataManagerSpec

Record getProperties() const

See also:

PlasmaStMan::getProperties

void setProperties(const Record &props)

See also:

PlasmaStMan::setProperties

inline rownr_t nrows() const

Return the number of rows used by all columns managed by this storage manager

Returns

The number of rows used by all columns managed by this storage manager

Public Static Functions

static DataManager *makeObject(const String &aDataManType, const Record &spec)

See also:

PlasmaStMan::makeObject

class PlasmaStManColumn : public StManColumnBase

A single column of the Plasma Storage Manager

A *PlasmaStManColumn* manages a single column on a casacore Table, which will be backed up by an Arrow object stored in Plasma. The actual handling of the underlying Arrow object is done via an *ArrowReader* instace, which hides the differences between the different types of Arrow objects that can hold data. At the moment the only supported reader is *TensorReader* (and thus this class still silently assumes that), but more will come. When the Tensor is retrieved from Plasma this class will create the corresponding *TensorReader* instance, which will ensure the data types are compatible. Also, upon data access (again, through the reader), the tensor's shape is compared against the column's cell shape to ensure the tensor and the column define the same dimensionality.

While casacore is column-major, Arrow is by default row-major. On the other hand, the dimensions that this column receives via setShapeColumn are those of individual cells, while Arrow Tensors will contain the full column data. Thus:

- The first dimension of the Tensor should always be the number of rows of the column
- For the rest of the dimensions, they should match the column cell's shape in reverse order.

In principle support for non-row-major Tensors should be possible to add, but that is left as a future improvement.

Public Functions

PlasmaStManColumn(const std::string &name, *PlasmaClient* &client, *PlasmaStMan*::*impl* &storage_manager, const ArrowObjectInfo &object_info, int dataType)

Create a new *PlasmaStManColumn* with the given name and data type. Upon construction it connects to Plasma and retrieves the underlying Arrow object, if known at this stage; otherwise a call to initialize_reader needs to be issued later before attempting to read anything.

Parameters

- **name** – The name of this column.
- **client** – The Plasma client object used to read Arrow objects off Plasma.
- **storage_manager** – A reference to the owning storage manager, used to retrieve the number of rows after table creation.
- **object_info** – Structure containing the Object ID and type of Arrow object to read from Plasma. If the type is ArrowObjectType::UNKNOWN then no reading occurs.
- **dataType** – The data type of this column.

void **initialize_reader**(const ArrowObjectInfo &object_info)

Initializes the underlying reader object with the provided information.

Parameters

object_info – Structure containing the Object ID and type of Arrow object to read from Plasma. If the type is ArrowObjectType::UNKNOWN then no initialization occurs.

bool **reader_initialized**() const

Returns

Whether the underlying reader is initialized or not.

4.2 Plasma access

class **PlasmaClient**

A class encapsulating access to a Plasma Store.

This class encapsulates access to a Plasma Store. Although it's a very thin wrapper around ::plasma::PlasmaClient, it adds configuration capabilities around certain aspects, like timeouts, the socket to connect to, retries and others.

Public Functions

PlasmaClient(std::string socket)

Create a new *PlasmaClient* that will connect to the given socket.

Parameters
socket – The Plasma socket to connect to.

void **ping**()

Ensure communication between the client and the server works.

inline void **set_get_timeout**(std::int64_t timeout)

Set the timeout for the Plasma Get operation, in milliseconds.

Parameters
timeout – The timeout for the Plasma Get operation, in milliseconds.

inline std::int64_t **get_timeout**() const

Returns
The timeout for the Plasma Get operation, in milliseconds.

inline void **set_connect_retries**(int connect_retries)

Set the number of attempts to connect to the Plasma socket before failing.

Parameters
connect_retries – the number of attempts to connect to the Plasma socket before failing.

inline int **connect_retries**() const

Returns
The number of attempts to connect to the Plasma socket before failing.

::plasma::ObjectBuffer **get**(const *ObjectID* &object_id)

Read an object from the Plasma store. A plasma_error exception is thrown if no such object is found within the timeout.

Parameters
object_id – The ID of the object to read.

Returns
A Plasma Object Buffer pointing to the object in the Plasma Store.

inline std::string **socket**() const

Returns
The socket where this Plasma client connects to.

4.3 Data reading

Internally, data reading is organised in a hierarchy of the *Reader* classes, each taking care of reading different Arrow objects.

class **ArrowReader**

Base class for Arrow data readers used by the *PlasmaStManColumn* class.

Arrow offers different storage types, like Tensors and Tables. This base class offers a common interface for accessing data from these different storage types.

Subclassed by *ska::plasma::TableReader*, *ska::plasma::TensorReader*

Public Functions

inline **ArrowReader**(const std::string &column_name, casacore::DataType data_type)

Constructs a reader for the given data type.

Parameters

- **column_name** – The casacore column backed by this reader.
- **data_type** – The casacore data type of the column backed by this reader.

virtual **~ArrowReader**() = default

Virtual destructor required by virtual base class.

inline void **check_conformance**(const Shape &column_shape)

Checks that the data type and the shape of the underlying Arrow object match those of the casacore column this reader backs up. The column data type is known at construction time, and the column shape is given here.

Parameters

column_shape – The shape of the casacore column this reader backs up.

virtual void **read_scalar**(rownr_t rownr, void *dataPtr) = 0

Read a single scalar value from the underlying Arrow object. The scalar value is that corresponding to the cell in row *rownr*.

Parameters

- **rownr** – The (casacore) row number of the cell for which the scalar is being read.
- **dataPtr** – The address where the scalar should be written to.

virtual void **read_array**(ArrayBase &array, std::size_t offset) = 0

Read an array from the underlying Arrow object starting at the given offset. The array's shape determines how much data is effectively read, and might or might not be able to be created with zero-copy.

Parameters

- **array** – The array where the data should be read into.
- **offset** – The offset in the underlying Arrow object at which reading will start.

class **TensorReader** : public ska::plasma::*ArrowReader*

An *ArrowReader* that reads data off an Arrow Tensor.

TODO: The current implementation contains two private templated methods to handle all data types. This means we need to continuously do a runtime check for the casacore data type to choose the correct template instance. This could be avoided by offering a TensorReaderBase class that handles all common aspects, then a *TensorReader* class templated on the casacore data type, and finally a factory function that is called once from *PlasmaStManColumn* to create the correct reader for the given casacore data type.

Public Functions

TensorReader(const std::string &column_name, casacore::DataType data_type, arrow::io::InputStream *input_stream)

Constructs a *TensorReader* for the given casacore data type and column from an input stream.

Parameters

- **column_name** – The casacore column backed by this reader.
- **data_type** – The casacore data type of the column backed by this
- **input_stream** – The input stream from where the Tensor will be read. This is possibly created from an object read from Plasma.

virtual void **read_scalar**(rownr_t rownr, void *dataPtr) override

Read a single scalar value from the underlying Arrow object. The scalar value is that corresponding to the cell in row *rownr*.

Parameters

- **rownr** – The (casacore) row number of the cell for which the scalar is being read.
- **dataPtr** – The address where the scalar should be written to.

virtual void **read_array**(ArrayBase &array, std::size_t offset) override

Read an array from the underlying Arrow object starting at the given offset. The array's shape determines how much data is effectively read, and might or might not be able to be created with zero-copy.

Parameters

- **array** – The array where the data should be read into.
- **offset** – The offset in the underlying Arrow object at which reading will start.

class **TableReader** : public ska::plasma::*ArrowReader*

An *ArrowReader* that reads data off an Arrow Table.

Tables can contain multiple "fields" or "columns". The column read by this reader is the one with the same name of the casacore Table column backed up by this reader. If no such field/column is found in the Arrow Table then an error is raised. Only Tables written as a single BatchRecord are currently supported.

Public Functions

TableReader(const std::string &column_name, casacore::DataType data_type, arrow::io::InputStream *input_stream)

Constructs a *TableReader* for the given casacore data type and column from an input stream. The column name in casacore must be the same as the column in the Arrow Table that will be read.

Parameters

- **column_name** – The casacore column backed by this reader. Should be the same as the column in the Arrow Table.
- **data_type** – The casacore data type of the column backed by this
- **input_stream** – The input stream from where the Table will be read. This is possibly created from an object read from Plasma.

virtual void **read_scalar**(rownr_t rownr, void *dataPtr) override

Read a single scalar value from the underlying Arrow object. The scalar value is that corresponding to the cell in row *rownr*.

Parameters

- **rownr** – The (casacore) row number of the cell for which the scalar is being read.
- **dataPtr** – The address where the scalar should be written to.

virtual void **read_array**(ArrayBase &array, std::size_t offset) override

Read an array from the underlying Arrow object starting at the given offset. The array's shape determines how much data is effectively read, and might or might not be able to be created with zero-copy.

Parameters

- **array** – The array where the data should be read into.
- **offset** – The offset in the underlying Arrow object at which reading will start.

4.4 Misc

class **ObjectID**

Simple, immutable class containing an Object ID.

This is a simpler version of plasma's own Object ID class, but without carrying all its dependencies, allowing us to have a specific type to represent Object IDs (other than std::string) without permeating the codebase with plasma dependencies.

Public Functions

ObjectID() = default

Construct an empty *ObjectID*, it can't be used for anything.

ObjectID(const std::string &object_id)

Constructs an Object ID for the given string, which must be a valid plasma Object ID.

Parameters
object_id – The contents of the Object ID

ObjectID(const char *object_id)

Constructs an Object ID for the given null-terminated C string, which must be a valid plasma Object ID.

Parameters
object_id – The contents of the Object ID

inline const std::string &**string**() const

Returns the underlying string.

Returns
The underlying string

inline bool **valid**() const

Returns whether this is a valid Object ID or not.

Returns
true if this Object ID is valid

INDEX

S