cuDF internals — cudf 0.19.0 documentation

The cuDF API closely matches that of the Pandas library. Thus, we have the types `cudf.Series`, `cudf.DataFrame` and `cudf.Index` which look and feel very much like their Pandas counterparts.

Under the hood, however, cuDF uses data structures very different from Pandas. In this document, we describe these internal data structures.

Column¶

Columns are cuDF's core data structure and they are modeled after the Apache Arrow Columnar Format.

A column represents a sequence of values, any number of which may be "null". Columns are specialized based on the type of data they contain. Thus we have `NumericalColumn`, `StringColumn`, `DatetimeColumn`, etc.,

A column is composed of the following:

- A **data type**, specifying the type of each element.
- A **data buffer** that may store the data for the column elements. Some column types do not have a data buffer, instead storing data in the children columns.
- A **mask buffer** whose bits represent the validity (null or not null) of each element. Columns whose elements are all "valid" may not have a mask buffer. Mask buffers are padded to 64 bytes.
- A tuple of **children** columns, which enable the representation complex types such as columns with non-fixed width elements such as strings or lists.
- A **size** indicating the number of elements in the column.
- An integer **offset**: a column may represent a "slice" of another column, in which case this offset represents the first element of the slice. The size of the column then gives the extent of the slice. A column that is not a slice has an offset of 0.

For example, the `NumericalColumn` backing a Series with 1000 elements of type 'int32' and containing nulls is composed of:

1. A data buffer of size 4000 bytes (sizeof(int32) * 1000)
2. A mask buffer of size 128 bytes (1000/8 padded to a multiple of 64 bytes)
3. No children columns

As another example, the `StringColumn` backing the Series `['do', 'you', 'have', 'any', 'cheese?']` is composed of:

1. No data buffer
2. No mask buffer as there are no nulls in the Series
3. Two children columns:
    - A column of 8-bit characters `['d', 'o', 'y', 'o', 'u', h' ... '?']`
    - A column of "offsets" to the characters column (in this case, `[0, 2, 5, 9, 12, 19]`)

Buffer¶

The data and mask buffers of a column represent data in GPU memory (a.k.a *device memory*), and are object of type `cudf.core.buffer.Buffer`.

Buffers can be constructed from array-like objects that live either on the host (e.g., numpy arrays) or the device (e.g., cupy arrays). Arrays must be of `uint8` dtype or viewed as such.

When constructing a Buffer from a host object such as a numpy array, new device memory is allocated:

```
>>> from cudf.core.buffer import Buffer
>>> buf = Buffer(np.array([1, 2, 3], dtype='int64').view('uint8'))
>>> print(buf.ptr)  # address of new device memory allocation
140050901762560
>>> print(buf.size)
24
>>> print(buf._owner)
<rmm._lib.device_buffer.DeviceBuffer object at 0x7f6055baab50>
```

cuDF uses the RMM library for allocating device memory. You can read more about device memory allocation with RMM here.

When constructing a Buffer from a device object such as a CuPy array, no new device memory is allocated. Instead, the Buffer points to the existing allocation, keeping a reference to the device array:

```
>>> import cupy as cp
>>> c_ary = cp.asarray([1, 2, 3], dtype='int64')
>>> buf = Buffer(c_ary.view('uint8'))
>>> print(c_ary.data.mem.ptr)
140050901762560
>>> print(buf.ptr)
140050901762560
>>> print(buf.size)
24
>>> print(buf._owner is c_ary)
True
```

An uninitialized block of device memory can be allocated with `Buffer.empty`:

```
>>> buf = Buffer.empty(10)
>>> print(buf.size)
10
>>> print(buf._owner)
<rmm._lib.device_buffer.DeviceBuffer object at 0x7f6055baa890>
```

ColumnAccessor¶

cuDF `Series`, `DataFrame` and `Index` are all subclasses of an internal `Frame` class. The underlying data structure of `Frame` is an ordered, dictionary-like object known as `ColumnAccessor`, which can be accessed via the `._data` attribute:

```
>>> a = cudf.DataFrame({'x': [1, 2, 3], 'y': ['a', 'b', 'c']})
>>> a._data
ColumnAccessor(OrderedColumnDict([('x', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d12e050>), ('y', <cudf.core.column.string.StringColumn object at 0x7f5a7d12e320>)]),
multiindex=False, level_names=(None,))
```

ColumnAccessor is an ordered mapping of column labels to columns. In addition to behaving like an OrderedDict, it supports things like selecting multiple columns (both by index and label), as well as hierarchical indexing.

```
>>> from cudf.core.column_accessor import ColumnAccessor
```

The values of a ColumnAccessor are coerced to Columns during construction:

```
>>> ca = ColumnAccessor({'x': [1, 2, 3], 'y': ['a', 'b', 'c']})
>>> ca['x']
<cudf.core.column.numerical.NumericalColumn object at 0x7f5a7d5789e0>
>>> ca['y']
<cudf.core.column.string.StringColumn object at 0x7f5a7d578b90>
>>> ca.pop('x')
<cudf.core.column.numerical.NumericalColumn object at 0x7f5a7d5789e0>
>>> ca
ColumnAccessor(OrderedColumnDict([('y', <cudf.core.column.string.StringColumn object at
0x7f5a7d578b90>)]), multiindex=False, level_names=(None,))
```

Columns can be inserted at a specified location:

```
>>> ca.insert('z', [3, 4, 5], loc=1)
>>> ca
ColumnAccessor(OrderedColumnDict([('x', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d578dd0>), ('z', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d578680>), ('y', <cudf.core.column.string.StringColumn object at 0x7f5a7d12e3b0>)]),
multiindex=False, level_names=(None,))
```

Selecting columns by index:

```
>>> ca = ColumnAccessor({'x': [1, 2, 3], 'y': ['a', 'b', 'c'], 'z': [4, 5, 6]})
>>> ca.select_by_index(1)
ColumnAccessor(OrderedColumnDict([('y', <cudf.core.column.string.StringColumn object at
0x7f5a7d578830>)]), multiindex=False, level_names=(None,))
>>> ca.select_by_index([0, 1])
ColumnAccessor(OrderedColumnDict([('x', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d5789e0>), ('y', <cudf.core.column.string.StringColumn object at 0x7f5a7d578830>)]),
multiindex=False, level_names=(None,))
>>> ca.select_by_index(slice(1, 3))
ColumnAccessor(OrderedColumnDict([('y', <cudf.core.column.string.StringColumn object at
0x7f5a7d578830>), ('z', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d5788c0>)]), multiindex=False, level_names=(None,))
```

Selecting columns by label:

```
>>> ca.select_by_label(['y', 'z'])
ColumnAccessor(OrderedColumnDict([('y', <cudf.core.column.string.StringColumn object at
0x7f5a7d578830>), ('z', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d5788c0>)]), multiindex=False, level_names=(None,))
>>> ca.select_by_label(slice('x', 'y'))
ColumnAccessor(OrderedColumnDict([('x', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d5789e0>), ('y', <cudf.core.column.string.StringColumn object at 0x7f5a7d578830>)]),
multiindex=False, level_names=(None,))
```

A ColumnAccessor with tuple keys (and constructed with `multiindex=True`) can be hierarchically indexed:

```
>>> ca = ColumnAccessor({('a', 'b'): [1, 2, 3], ('a', 'c'): [2, 3, 4], 'b': [4, 5, 6]},
multiindex=True)
>>> ca.select_by_label('a')
ColumnAccessor(OrderedColumnDict([('b', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d5789e0>), ('c', <cudf.core.column.numerical.NumericalColumn object at
0x7f5a7d578dd0>)]), multiindex=False, level_names=(None,))
>>> ca.select_by_label(('a', 'b'))
ColumnAccessor(OrderedColumnDict([(('a', 'b'), <cudf.core.column.numerical.NumericalColumn
object at 0x7f5a7d5789e0>)]), multiindex=False, level_names=(None,))
```

"Wildcard" indexing is also allowed:

```
>>> ca = ColumnAccessor({('a', 'b'): [1, 2, 3], ('a', 'c'): [2, 3, 4], ('d', 'b'): [4, 5, 6]},
multiindex=True)
>>> ca.select_by_label((slice(None), 'b'))
ColumnAccessor(OrderedColumnDict([(('a', 'b'), <cudf.core.column.numerical.NumericalColumn
object at 0x7f5a7d578830>), (('d', 'b'), <cudf.core.column.numerical.NumericalColumn object
at 0x7f5a7d578680>)]), multiindex=True, level_names=(None, None))
```

Finally, ColumnAccessors can convert to Pandas `Index` or `MultiIndex` objects:

```
>>> ca.to_pandas_index()
MultiIndex([('a', 'b'),
            ('a', 'c'),
            ('d', 'b')],
           )
```