Lookout, nested types are now supported in RAPIDS cuDF!

https://medium.com/rapids-ai/lookout-nested-types-are-now-supported-in-rapids-cudf-c6e138b288e8

Ashwin Srinath Wed Jun, 09 17:16

RAPIDS cuDF release 0.19 introduces improved support for *nested* types such as lists and structs. This means that you can now leverage the GPU to perform operations on even more complex data while using familiar Pandas-like syntax.

```
>>> import cudf
>>> cudf.Series([[1, 2, 3], [4, 5], [6, 7]])
>>> s
Out[9]:
0 [1, 2, 3]
1 [4, 5]
2 [6, 7]
dtype: list
>>> s.dtype
ListDtype(int64) # a new dtype!
```

This blog post will cover everything you need to know about nested types in cuDF: what they are, how they work, and how they can be used.

In Pandas, you can create a Series containing almost any kind of value. This includes lists, dictionaries, or even your own custom type. The dtype for such a Series is `object`, representing a collection of arbitrary Python objects. Operating on a Series of type `object` is often slow compared to the same operation on more specific types. Consider the difference in performance when computing the `max` of two Series objects with dtype `int64` and `object` respectively:

```
s = pd.Series(range(100_000), dtype="int64')%timeit s.max()
106 µs ± 205 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)s =
pd.Series(range(100_000), dtype="object")%timeit s.max()
9.7 ms ± 6.84 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The same operation on the `object` Series is almost 100 times slower! This is because the numpy array backing this Series behaves more like a list, storing references to several individual Python objects rather than a single contiguous region in memory. Without knowing any more about the data than "it's a collection of Python objects", NumPy cannot perform operations very efficiently on it.

In cuDF, we don't have a general `object` dtype. This means that unlike with Pandas, you can't put anything you like inside a cuDF Series. However, we *do* have a few specialized dtypes to support the most common use cases. These include:

1. A ListDtype for Series of list-like values (or lists of lists, or…)

2. A StructDtype for Series of struct values (or struct of struct, or…). A *struct* is an ordered mapping of *field names* to values.

These "nested" types blend flexibility and efficiency by enabling you to store complex data and operate on it using the GPU.

You can create lists and struct Series from existing Pandas Series of lists and dictionaries respectively:

```
>>> psr = pd.Series([{'a': 1, 'b': 2}, {'a': 3, 'b': 4}])
>>> psr
0 {'a': 1, 'b': 2}
1 {'a': 3, 'b': 4}
dtype: object
>>> gsr = cudf.from_pandas(psr)
>>> gsr
0 {'a': 1, 'b': 2}
1 {'a': 3, 'b': 4}
dtype: struct
```

Or by reading them from disk, using a file format that supports nested data. Currently, only the Parquet format is supported, but the ORC format is currently being worked on and the JSON and Avro formats are planned for the future as well.

```
>>> pdf = pd.DataFrame({"a": [{'a': 1, 'b': 2}, {'a': 3, 'b': 4}]})
>>> pdf
a
0 {'a': 1, 'b': 2}
1 {'a': 3, 'b': 4}
>>> pdf.to_parquet("example.parquet")
>>> cudf.read_parquet("struct.pq")
a
0 {'a': 1, 'b': 2}
1 {'a': 3, 'b': 4}
```

General operations

Many simple operations on nested types "just work". Slicing, selecting, masking, and concatenating all work as expected:

```
>>> s
0 [1, 2, 3]
1 [4, 5]
2 [6, 7, 8]
3 [9, 10]
dtype: list
>>> s[[0, 2]]
0 [1, 2, 3]
2 [6, 7, 8]
dtype: list
>>> cudf.concat([s, s])
0 [1, 2, 3]
1 [4, 5]
2 [6, 7, 8]
3 [9, 10]
0 [1, 2, 3]
1 [4, 5]
2 [6, 7, 8]
3 [9, 10]
dtype: list
```

But more complex operations will likely not work today. In particular, nested types cannot currently be compared, so trying to sort a list or struct Series will produce an error.

Type-specific operations

In addition to these general operations, cuDF supports type-specific operations on nested types via accessors. Similar to the `.str` or `.dt` accessors for string- and datetime-specific operations, cuDF provides `.list` and `.struct` accessors:

Count the number of elements in each list in a list Series:

```
>>> x
0 [1, 2, 3]
1 [4, 5]
2 [7, 8, 9, 10]
dtype: list
>>> x.list.len()
Out[22]:
0 3
1 2
2 4
dtype: int32
```

Extract the values corresponding to a specific key in a struct Series:

```
>>> y
0 {'a': 1, 'b': 2}
1 {'a': 3, 'b': 4}
dtype: struct
>>> y.struct.field('b')
0 2
1 4
dtype: int64
```

How they work (internals)

cuDF uses the Apache Arrow Columnar Format. Under this specification, a list Series in cuDF is actually composed of two "children" columns:

1. A column containing the elements from each list in the list Series.
2. A column of offsets into the first column, indicating the start and end element of each row

This is easier to understand with an example. Consider the Series below:

```
>>> s
0 [1, 2, 3]
1 [4, 5]
2 [6, 7, 8]
3 [9, 10]
dtype: list
```

The `elements` column of this Series is `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`, and its `offsets` column is `[0, 3, 5, 8, 10]`.

A struct Series is simply composed of `n` children columns, `n` being the number of fields in each struct.

```
>>> s
0 {'a': 1, 'b': 2, 'c': 3}
1 {'a': 4, 'b': 5, 'c': 6}
2 {'a': 7, 'b': 8, 'c': 9}
dtype: struct
s._column.children
(<cudf.core.column.numerical.NumericalColumn object at 0x7f2f9373af80>
[
1,
4,
7]
dtype: int64,
<cudf.core.column.numerical.NumericalColumn object at 0x7f2f986483b0>
[
2,
5,
8
]
dtype: int64,
<cudf.core.column.numerical.NumericalColumn object at 0x7f2f935ea8c0>
[
3,
6,
9
]
dtype: int64)
```

You can read more about the internals of cuDF here.

Operations on nested types in cuDF can be significantly faster than the equivalent operations on objects in Pandas.

Sorting each row

```
%timeit pandas_sr.apply(sorted)
355 ms ± 298 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)%timeit
cudf_sr.list.sort_values()
9.56 ms ± 39 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Computing the unique values in each row

```
%timeit pandas_sr.apply(set)
51 ms ± 516 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)%timeit
cudf_sr.list.unique()
10.3 ms ± 284 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Note: *the measurements above were done using cuDF 0.20*

Nested types are a new feature in cuDF. They also represent one of our first attempts to grow *beyond* the Pandas API. Do you have a use case that can benefit from nested types? Is there an operation that you wish cuDF would support on nested types? Would you like to contribute a feature or documentation? If so, start a discussion by raising an issue on our GitHub repository. We'd love to hear from you!