Use with PyArrow

This document is a quick introduction to using `datasets` with PyArrow, with a particular focus on how to process
datasets using Arrow compute functions, and how to convert a dataset to PyArrow or from PyArrow.

This is particularly useful as it allows fast zero-copy operations, since `datasets` uses PyArrow under the hood.

Dataset format

By default, datasets return regular Python objects: integers, floats, strings, lists, etc.

To get PyArrow Tables or Arrays instead, you can set the format of the dataset to `pyarrow` using Dataset.with_format():

```
>>> from datasets import Dataset
>>> data = {"col_0": ["a", "b", "c", "d"], "col_1": [0., 0., 1., 1.]}
>>> ds = Dataset.from_dict(data)
>>> ds = ds.with_format("arrow")
>>> ds[0]       # pa.Table
pyarrow.Table
col_0: string
col_1: double
----
col_0: [["a"]]
col_1: [[0]]
>>> ds[:2]      # pa.Table
pyarrow.Table
col_0: string
col_1: double
----
col_0: [["a","b"]]
col_1: [[0,0]]
>>> ds["data"]  # pa.array
<pyarrow.lib.ChunkedArray object at 0x1394312a0>
[
  [
    "a",
    "b",
    "c",
    "d"
  ]
]
```

This also works for `IterableDataset` objects obtained e.g. using `load_dataset(..., streaming=True)`:

```
>>> ds = ds.with_format("arrow")
>>> for table in ds.iter(batch_size=2):
...     print(table)
...     break
pyarrow.Table
col_0: string
col_1: double
----
col_0: [["a","b"]]
col_1: [[0,0]]
```

Process data

PyArrow functions are generally faster than regular hand-written python functions, and therefore they are a good option to optimize data processing. You can use Arrow compute functions to process a dataset in Dataset.map() or Dataset.filter():

```
>>> import pyarrow.compute as pc
>>> from datasets import Dataset
>>> data = {"col_0": ["a", "b", "c", "d"], "col_1": [0., 0., 1., 1.]}
>>> ds = Dataset.from_dict(data)
>>> ds = ds.with_format("arrow")
>>> ds = ds.map(lambda t: t.append_column("col_2", pc.add(t["col_1"], 1)), batched=True)
>>> ds[:2]
pyarrow.Table
col_0: string
col_1: double
col_2: double
----
col_0: [["a","b"]]
col_1: [[0,0]]
col_2: [[1,1]]
>>> ds = ds.filter(lambda t: pc.equal(t["col_0"], "b"), batched=True)
>>> ds[0]
pyarrow.Table
col_0: string
col_1: double
col_2: double
----
col_0: [["b"]]
col_1: [[0]]
col_2: [[1]]
```

We use `batched=True` because it is faster to process batches of data in PyArrow rather than row by row. It's also possible to use `batch_size=` in `map()` to set the size of each `table`.

This also works for IterableDataset.map() and IterableDataset.filter().

Import or Export from PyArrow

A Dataset is a wrapper of a PyArrow Table, you can instantiate a Dataset directly from the Table:

```
ds = Dataset(table)
```

You can access the PyArrow Table of a dataset using Dataset.data, which returns a `MemoryMappedTable` or a `InMemoryTable` or a `ConcatenationTable`, depending on the origin of the Arrow data and the operations that were applied.

Those objects wrap the underlying PyArrow table accessible at `Dataset.data.table`. This table contains all the data of the dataset, but there might also be an indices mapping at `Dataset._indices` which maps the dataset rows indices to the PyArrow Table rows indices. This can happen if the dataset has been shuffled with Dataset.shuffle() or if only a subset of the rows are used (e.g. after a Dataset.select()).

In the general case, you can export a dataset to a PyArrow Table using `table = ds.with_format("arrow")[:]`.