// Licensed to the Apache Software Foundation (ASF) under one

// or more contributor license agreements. See the NOTICE file

// distributed with this work for additional information

// regarding copyright ownership. The ASF licenses this file

// to you under the Apache License, Version 2.0 (the

// "License"); you may not use this file except in compliance

// with the License. You may obtain a copy of the License at

//

// http://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing,

// software distributed under the License is distributed on an

// "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

// KIND, either express or implied. See the License for the

// specific language governing permissions and limitations

// under the License.

#include "arrow/python/numpy\_interop.h"

#include "arrow/python/numpy\_convert.h"

#include <cstdint>

#include <memory>

#include <string>

#include <vector>

#include "arrow/buffer.h"

#include "arrow/sparse\_tensor.h"

#include "arrow/tensor.h"

#include "arrow/type.h"

#include "arrow/util/logging.h"

#include "arrow/python/common.h"

#include "arrow/python/pyarrow.h"

#include "arrow/python/type\_traits.h"

namespace arrow {

namespace py {

NumPyBuffer::NumPyBuffer(PyObject\* ao) : Buffer(nullptr, 0) {

PyAcquireGIL lock;

arr\_ = ao;

Py\_INCREF(ao);

if (PyArray\_Check(ao)) {

PyArrayObject\* ndarray = reinterpret\_cast<PyArrayObject\*>(ao);

auto ptr = reinterpret\_cast<uint8\_t\*>(PyArray\_DATA(ndarray));

data\_ = const\_cast<const uint8\_t\*>(ptr);

size\_ = PyArray\_NBYTES(ndarray);

capacity\_ = size\_;

is\_mutable\_ = !!(PyArray\_FLAGS(ndarray) & NPY\_ARRAY\_WRITEABLE);

}

}

NumPyBuffer::~NumPyBuffer() {

PyAcquireGIL lock;

Py\_XDECREF(arr\_);

}

#define TO\_ARROW\_TYPE\_CASE(NPY\_NAME, FACTORY) \

case NPY\_##NPY\_NAME: \

return FACTORY();

namespace {

Result<std::shared\_ptr<DataType>> GetTensorType(PyObject\* dtype) {

if (!PyObject\_TypeCheck(dtype, &PyArrayDescr\_Type)) {

return Status::TypeError("Did not pass numpy.dtype object");

}

PyArray\_Descr\* descr = reinterpret\_cast<PyArray\_Descr\*>(dtype);

int type\_num = fix\_numpy\_type\_num(descr->type\_num);

switch (type\_num) {

TO\_ARROW\_TYPE\_CASE(BOOL, uint8);

TO\_ARROW\_TYPE\_CASE(INT8, int8);

TO\_ARROW\_TYPE\_CASE(INT16, int16);

TO\_ARROW\_TYPE\_CASE(INT32, int32);

TO\_ARROW\_TYPE\_CASE(INT64, int64);

TO\_ARROW\_TYPE\_CASE(UINT8, uint8);

TO\_ARROW\_TYPE\_CASE(UINT16, uint16);

TO\_ARROW\_TYPE\_CASE(UINT32, uint32);

TO\_ARROW\_TYPE\_CASE(UINT64, uint64);

TO\_ARROW\_TYPE\_CASE(FLOAT16, float16);

TO\_ARROW\_TYPE\_CASE(FLOAT32, float32);

TO\_ARROW\_TYPE\_CASE(FLOAT64, float64);

}

return Status::NotImplemented("Unsupported numpy type ", descr->type\_num);

}

Status GetNumPyType(const DataType& type, int\* type\_num) {

#define NUMPY\_TYPE\_CASE(ARROW\_NAME, NPY\_NAME) \

case Type::ARROW\_NAME: \

\*type\_num = NPY\_##NPY\_NAME; \

break;

switch (type.id()) {

NUMPY\_TYPE\_CASE(UINT8, UINT8);

NUMPY\_TYPE\_CASE(INT8, INT8);

NUMPY\_TYPE\_CASE(UINT16, UINT16);

NUMPY\_TYPE\_CASE(INT16, INT16);

NUMPY\_TYPE\_CASE(UINT32, UINT32);

NUMPY\_TYPE\_CASE(INT32, INT32);

NUMPY\_TYPE\_CASE(UINT64, UINT64);

NUMPY\_TYPE\_CASE(INT64, INT64);

NUMPY\_TYPE\_CASE(HALF\_FLOAT, FLOAT16);

NUMPY\_TYPE\_CASE(FLOAT, FLOAT32);

NUMPY\_TYPE\_CASE(DOUBLE, FLOAT64);

default: {

return Status::NotImplemented("Unsupported tensor type: ", type.ToString());

}

}

#undef NUMPY\_TYPE\_CASE

return Status::OK();

}

} // namespace

Result<std::shared\_ptr<DataType>> NumPyScalarToArrowDataType(PyObject\* scalar) {

PyArray\_Descr\* descr = PyArray\_DescrFromScalar(scalar);

OwnedRef descr\_ref(reinterpret\_cast<PyObject\*>(descr));

return NumPyDtypeToArrow(descr);

}

Result<std::shared\_ptr<DataType>> NumPyDtypeToArrow(PyObject\* dtype) {

if (!PyObject\_TypeCheck(dtype, &PyArrayDescr\_Type)) {

return Status::TypeError("Did not pass numpy.dtype object");

}

PyArray\_Descr\* descr = reinterpret\_cast<PyArray\_Descr\*>(dtype);

return NumPyDtypeToArrow(descr);

}

Result<std::shared\_ptr<DataType>> NumPyDtypeToArrow(PyArray\_Descr\* descr) {

int type\_num = fix\_numpy\_type\_num(descr->type\_num);

switch (type\_num) {

TO\_ARROW\_TYPE\_CASE(BOOL, boolean);

TO\_ARROW\_TYPE\_CASE(INT8, int8);

TO\_ARROW\_TYPE\_CASE(INT16, int16);

TO\_ARROW\_TYPE\_CASE(INT32, int32);

TO\_ARROW\_TYPE\_CASE(INT64, int64);

TO\_ARROW\_TYPE\_CASE(UINT8, uint8);

TO\_ARROW\_TYPE\_CASE(UINT16, uint16);

TO\_ARROW\_TYPE\_CASE(UINT32, uint32);

TO\_ARROW\_TYPE\_CASE(UINT64, uint64);

TO\_ARROW\_TYPE\_CASE(FLOAT16, float16);

TO\_ARROW\_TYPE\_CASE(FLOAT32, float32);

TO\_ARROW\_TYPE\_CASE(FLOAT64, float64);

TO\_ARROW\_TYPE\_CASE(STRING, binary);

TO\_ARROW\_TYPE\_CASE(UNICODE, utf8);

case NPY\_DATETIME: {

auto date\_dtype =

reinterpret\_cast<PyArray\_DatetimeDTypeMetaData\*>(PyDataType\_C\_METADATA(descr));

switch (date\_dtype->meta.base) {

case NPY\_FR\_s:

return timestamp(TimeUnit::SECOND);

case NPY\_FR\_ms:

return timestamp(TimeUnit::MILLI);

case NPY\_FR\_us:

return timestamp(TimeUnit::MICRO);

case NPY\_FR\_ns:

return timestamp(TimeUnit::NANO);

case NPY\_FR\_D:

return date32();

case NPY\_FR\_GENERIC:

return Status::NotImplemented("Unbound or generic datetime64 time unit");

default:

return Status::NotImplemented("Unsupported datetime64 time unit");

}

} break;

case NPY\_TIMEDELTA: {

auto timedelta\_dtype =

reinterpret\_cast<PyArray\_DatetimeDTypeMetaData\*>(PyDataType\_C\_METADATA(descr));

switch (timedelta\_dtype->meta.base) {

case NPY\_FR\_s:

return duration(TimeUnit::SECOND);

case NPY\_FR\_ms:

return duration(TimeUnit::MILLI);

case NPY\_FR\_us:

return duration(TimeUnit::MICRO);

case NPY\_FR\_ns:

return duration(TimeUnit::NANO);

case NPY\_FR\_GENERIC:

return Status::NotImplemented("Unbound or generic timedelta64 time unit");

default:

return Status::NotImplemented("Unsupported timedelta64 time unit");

}

} break;

}

return Status::NotImplemented("Unsupported numpy type ", descr->type\_num);

}

#undef TO\_ARROW\_TYPE\_CASE

Status NdarrayToTensor(MemoryPool\* pool, PyObject\* ao,

const std::vector<std::string>& dim\_names,

std::shared\_ptr<Tensor>\* out) {

if (!PyArray\_Check(ao)) {

return Status::TypeError("Did not pass ndarray object");

}

PyArrayObject\* ndarray = reinterpret\_cast<PyArrayObject\*>(ao);

// TODO(wesm): What do we want to do with non-contiguous memory and negative strides?

int ndim = PyArray\_NDIM(ndarray);

std::shared\_ptr<Buffer> data = std::make\_shared<NumPyBuffer>(ao);

std::vector<int64\_t> shape(ndim);

std::vector<int64\_t> strides(ndim);

npy\_intp\* array\_strides = PyArray\_STRIDES(ndarray);

npy\_intp\* array\_shape = PyArray\_SHAPE(ndarray);

for (int i = 0; i < ndim; ++i) {

if (array\_strides[i] < 0) {

return Status::Invalid("Negative ndarray strides not supported");

}

shape[i] = array\_shape[i];

strides[i] = array\_strides[i];

}

ARROW\_ASSIGN\_OR\_RAISE(

auto type, GetTensorType(reinterpret\_cast<PyObject\*>(PyArray\_DESCR(ndarray))));

\*out = std::make\_shared<Tensor>(type, data, shape, strides, dim\_names);

return Status::OK();

}

Status TensorToNdarray(const std::shared\_ptr<Tensor>& tensor, PyObject\* base,

PyObject\*\* out) {

int type\_num = 0;

RETURN\_NOT\_OK(GetNumPyType(\*tensor->type(), &type\_num));

PyArray\_Descr\* dtype = PyArray\_DescrNewFromType(type\_num);

RETURN\_IF\_PYERROR();

const int ndim = tensor->ndim();

std::vector<npy\_intp> npy\_shape(ndim);

std::vector<npy\_intp> npy\_strides(ndim);

for (int i = 0; i < ndim; ++i) {

npy\_shape[i] = tensor->shape()[i];

npy\_strides[i] = tensor->strides()[i];

}

const void\* immutable\_data = nullptr;

if (tensor->data()) {

immutable\_data = tensor->data()->data();

}

// Remove const =(

void\* mutable\_data = const\_cast<void\*>(immutable\_data);

int array\_flags = 0;

if (tensor->is\_row\_major()) {

array\_flags |= NPY\_ARRAY\_C\_CONTIGUOUS;

}

if (tensor->is\_column\_major()) {

array\_flags |= NPY\_ARRAY\_F\_CONTIGUOUS;

}

if (tensor->is\_mutable()) {

array\_flags |= NPY\_ARRAY\_WRITEABLE;

}

PyObject\* result =

PyArray\_NewFromDescr(&PyArray\_Type, dtype, ndim, npy\_shape.data(),

npy\_strides.data(), mutable\_data, array\_flags, nullptr);

RETURN\_IF\_PYERROR();

if (base == Py\_None || base == nullptr) {

base = py::wrap\_tensor(tensor);

} else {

Py\_XINCREF(base);

}

PyArray\_SetBaseObject(reinterpret\_cast<PyArrayObject\*>(result), base);

\*out = result;

return Status::OK();

}

// Wrap the dense data of a sparse tensor in a ndarray

static Status SparseTensorDataToNdarray(const SparseTensor& sparse\_tensor,

std::vector<npy\_intp> data\_shape, PyObject\* base,

PyObject\*\* out\_data) {

int type\_num\_data = 0;

RETURN\_NOT\_OK(GetNumPyType(\*sparse\_tensor.type(), &type\_num\_data));

PyArray\_Descr\* dtype\_data = PyArray\_DescrNewFromType(type\_num\_data);

RETURN\_IF\_PYERROR();

const void\* immutable\_data = sparse\_tensor.data()->data();

// Remove const =(

void\* mutable\_data = const\_cast<void\*>(immutable\_data);

int array\_flags = NPY\_ARRAY\_C\_CONTIGUOUS | NPY\_ARRAY\_F\_CONTIGUOUS;

if (sparse\_tensor.is\_mutable()) {

array\_flags |= NPY\_ARRAY\_WRITEABLE;

}

\*out\_data = PyArray\_NewFromDescr(&PyArray\_Type, dtype\_data,

static\_cast<int>(data\_shape.size()), data\_shape.data(),

nullptr, mutable\_data, array\_flags, nullptr);

RETURN\_IF\_PYERROR();

Py\_XINCREF(base);

PyArray\_SetBaseObject(reinterpret\_cast<PyArrayObject\*>(\*out\_data), base);

return Status::OK();

}

Status SparseCOOTensorToNdarray(const std::shared\_ptr<SparseCOOTensor>& sparse\_tensor,

PyObject\* base, PyObject\*\* out\_data,

PyObject\*\* out\_coords) {

const auto& sparse\_index = arrow::internal::checked\_cast<const SparseCOOIndex&>(

\*sparse\_tensor->sparse\_index());

// Wrap tensor data

OwnedRef result\_data;

RETURN\_NOT\_OK(SparseTensorDataToNdarray(

\*sparse\_tensor, {static\_cast<npy\_intp>(sparse\_tensor->non\_zero\_length()), 1}, base,

result\_data.ref()));

// Wrap indices

PyObject\* result\_coords;

RETURN\_NOT\_OK(TensorToNdarray(sparse\_index.indices(), base, &result\_coords));

\*out\_data = result\_data.detach();

\*out\_coords = result\_coords;

return Status::OK();

}

Status SparseCSXMatrixToNdarray(const std::shared\_ptr<SparseTensor>& sparse\_tensor,

PyObject\* base, PyObject\*\* out\_data,

PyObject\*\* out\_indptr, PyObject\*\* out\_indices) {

// Wrap indices

OwnedRef result\_indptr;

OwnedRef result\_indices;

switch (sparse\_tensor->format\_id()) {

case SparseTensorFormat::CSR: {

const auto& sparse\_index = arrow::internal::checked\_cast<const SparseCSRIndex&>(

\*sparse\_tensor->sparse\_index());

RETURN\_NOT\_OK(TensorToNdarray(sparse\_index.indptr(), base, result\_indptr.ref()));

RETURN\_NOT\_OK(TensorToNdarray(sparse\_index.indices(), base, result\_indices.ref()));

break;

}

case SparseTensorFormat::CSC: {

const auto& sparse\_index = arrow::internal::checked\_cast<const SparseCSCIndex&>(

\*sparse\_tensor->sparse\_index());

RETURN\_NOT\_OK(TensorToNdarray(sparse\_index.indptr(), base, result\_indptr.ref()));

RETURN\_NOT\_OK(TensorToNdarray(sparse\_index.indices(), base, result\_indices.ref()));

break;

}

default:

return Status::NotImplemented("Invalid SparseTensor type.");

}

// Wrap tensor data

OwnedRef result\_data;

RETURN\_NOT\_OK(SparseTensorDataToNdarray(

\*sparse\_tensor, {static\_cast<npy\_intp>(sparse\_tensor->non\_zero\_length()), 1}, base,

result\_data.ref()));

\*out\_data = result\_data.detach();

\*out\_indptr = result\_indptr.detach();

\*out\_indices = result\_indices.detach();

return Status::OK();

}

Status SparseCSRMatrixToNdarray(const std::shared\_ptr<SparseCSRMatrix>& sparse\_tensor,

PyObject\* base, PyObject\*\* out\_data,

PyObject\*\* out\_indptr, PyObject\*\* out\_indices) {

return SparseCSXMatrixToNdarray(sparse\_tensor, base, out\_data, out\_indptr, out\_indices);

}

Status SparseCSCMatrixToNdarray(const std::shared\_ptr<SparseCSCMatrix>& sparse\_tensor,

PyObject\* base, PyObject\*\* out\_data,

PyObject\*\* out\_indptr, PyObject\*\* out\_indices) {

return SparseCSXMatrixToNdarray(sparse\_tensor, base, out\_data, out\_indptr, out\_indices);

}

Status SparseCSFTensorToNdarray(const std::shared\_ptr<SparseCSFTensor>& sparse\_tensor,

PyObject\* base, PyObject\*\* out\_data,

PyObject\*\* out\_indptr, PyObject\*\* out\_indices) {

const auto& sparse\_index = arrow::internal::checked\_cast<const SparseCSFIndex&>(

\*sparse\_tensor->sparse\_index());

// Wrap tensor data

OwnedRef result\_data;

RETURN\_NOT\_OK(SparseTensorDataToNdarray(

\*sparse\_tensor, {static\_cast<npy\_intp>(sparse\_tensor->non\_zero\_length()), 1}, base,

result\_data.ref()));

// Wrap indices

int ndim = static\_cast<int>(sparse\_index.indices().size());

OwnedRef indptr(PyList\_New(ndim - 1));

OwnedRef indices(PyList\_New(ndim));

RETURN\_IF\_PYERROR();

for (int i = 0; i < ndim - 1; ++i) {

PyObject\* item;

RETURN\_NOT\_OK(TensorToNdarray(sparse\_index.indptr()[i], base, &item));

if (PyList\_SetItem(indptr.obj(), i, item) < 0) {

Py\_XDECREF(item);

RETURN\_IF\_PYERROR();

}

}

for (int i = 0; i < ndim; ++i) {

PyObject\* item;

RETURN\_NOT\_OK(TensorToNdarray(sparse\_index.indices()[i], base, &item));

if (PyList\_SetItem(indices.obj(), i, item) < 0) {

Py\_XDECREF(item);

RETURN\_IF\_PYERROR();

}

}

\*out\_indptr = indptr.detach();

\*out\_indices = indices.detach();

\*out\_data = result\_data.detach();

return Status::OK();

}

Status NdarraysToSparseCOOTensor(MemoryPool\* pool, PyObject\* data\_ao, PyObject\* coords\_ao,

const std::vector<int64\_t>& shape,

const std::vector<std::string>& dim\_names,

std::shared\_ptr<SparseCOOTensor>\* out) {

if (!PyArray\_Check(data\_ao) || !PyArray\_Check(coords\_ao)) {

return Status::TypeError("Did not pass ndarray object");

}

PyArrayObject\* ndarray\_data = reinterpret\_cast<PyArrayObject\*>(data\_ao);

std::shared\_ptr<Buffer> data = std::make\_shared<NumPyBuffer>(data\_ao);

ARROW\_ASSIGN\_OR\_RAISE(

auto type\_data,

GetTensorType(reinterpret\_cast<PyObject\*>(PyArray\_DESCR(ndarray\_data))));

std::shared\_ptr<Tensor> coords;

RETURN\_NOT\_OK(NdarrayToTensor(pool, coords\_ao, {}, &coords));

ARROW\_CHECK\_EQ(coords->type\_id(), Type::INT64); // Should be ensured by caller

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<SparseCOOIndex> sparse\_index,

SparseCOOIndex::Make(coords));

\*out = std::make\_shared<SparseTensorImpl<SparseCOOIndex>>(sparse\_index, type\_data, data,

shape, dim\_names);

return Status::OK();

}

template <class IndexType>

Status NdarraysToSparseCSXMatrix(MemoryPool\* pool, PyObject\* data\_ao, PyObject\* indptr\_ao,

PyObject\* indices\_ao, const std::vector<int64\_t>& shape,

const std::vector<std::string>& dim\_names,

std::shared\_ptr<SparseTensorImpl<IndexType>>\* out) {

if (!PyArray\_Check(data\_ao) || !PyArray\_Check(indptr\_ao) ||

!PyArray\_Check(indices\_ao)) {

return Status::TypeError("Did not pass ndarray object");

}

PyArrayObject\* ndarray\_data = reinterpret\_cast<PyArrayObject\*>(data\_ao);

std::shared\_ptr<Buffer> data = std::make\_shared<NumPyBuffer>(data\_ao);

ARROW\_ASSIGN\_OR\_RAISE(

auto type\_data,

GetTensorType(reinterpret\_cast<PyObject\*>(PyArray\_DESCR(ndarray\_data))));

std::shared\_ptr<Tensor> indptr, indices;

RETURN\_NOT\_OK(NdarrayToTensor(pool, indptr\_ao, {}, &indptr));

RETURN\_NOT\_OK(NdarrayToTensor(pool, indices\_ao, {}, &indices));

ARROW\_CHECK\_EQ(indptr->type\_id(), Type::INT64); // Should be ensured by caller

ARROW\_CHECK\_EQ(indices->type\_id(), Type::INT64); // Should be ensured by caller

auto sparse\_index = std::make\_shared<IndexType>(

std::static\_pointer\_cast<NumericTensor<Int64Type>>(indptr),

std::static\_pointer\_cast<NumericTensor<Int64Type>>(indices));

\*out = std::make\_shared<SparseTensorImpl<IndexType>>(sparse\_index, type\_data, data,

shape, dim\_names);

return Status::OK();

}

Status NdarraysToSparseCSFTensor(MemoryPool\* pool, PyObject\* data\_ao, PyObject\* indptr\_ao,

PyObject\* indices\_ao, const std::vector<int64\_t>& shape,

const std::vector<int64\_t>& axis\_order,

const std::vector<std::string>& dim\_names,

std::shared\_ptr<SparseCSFTensor>\* out) {

if (!PyArray\_Check(data\_ao)) {

return Status::TypeError("Did not pass ndarray object for data");

}

const int ndim = static\_cast<const int>(shape.size());

PyArrayObject\* ndarray\_data = reinterpret\_cast<PyArrayObject\*>(data\_ao);

std::shared\_ptr<Buffer> data = std::make\_shared<NumPyBuffer>(data\_ao);

ARROW\_ASSIGN\_OR\_RAISE(

auto type\_data,

GetTensorType(reinterpret\_cast<PyObject\*>(PyArray\_DESCR(ndarray\_data))));

std::vector<std::shared\_ptr<Tensor>> indptr(ndim - 1);

std::vector<std::shared\_ptr<Tensor>> indices(ndim);

for (int i = 0; i < ndim - 1; ++i) {

#ifdef Py\_GIL\_DISABLED

PyObject\* item = PySequence\_ITEM(indptr\_ao, i);

RETURN\_IF\_PYERROR();

OwnedRef item\_ref(item);

#else

PyObject\* item = PySequence\_Fast\_GET\_ITEM(indptr\_ao, i);

#endif

if (!PyArray\_Check(item)) {

return Status::TypeError("Did not pass ndarray object for indptr");

}

RETURN\_NOT\_OK(NdarrayToTensor(pool, item, {}, &indptr[i]));

ARROW\_CHECK\_EQ(indptr[i]->type\_id(), Type::INT64); // Should be ensured by caller

}

for (int i = 0; i < ndim; ++i) {

#ifdef Py\_GIL\_DISABLED

PyObject\* item = PySequence\_ITEM(indices\_ao, i);

RETURN\_IF\_PYERROR();

OwnedRef item\_ref(item);

#else

PyObject\* item = PySequence\_Fast\_GET\_ITEM(indices\_ao, i);

#endif

if (!PyArray\_Check(item)) {

return Status::TypeError("Did not pass ndarray object for indices");

}

RETURN\_NOT\_OK(NdarrayToTensor(pool, item, {}, &indices[i]));

ARROW\_CHECK\_EQ(indices[i]->type\_id(), Type::INT64); // Should be ensured by caller

}

auto sparse\_index = std::make\_shared<SparseCSFIndex>(indptr, indices, axis\_order);

\*out = std::make\_shared<SparseTensorImpl<SparseCSFIndex>>(sparse\_index, type\_data, data,

shape, dim\_names);

return Status::OK();

}

Status NdarraysToSparseCSRMatrix(MemoryPool\* pool, PyObject\* data\_ao, PyObject\* indptr\_ao,

PyObject\* indices\_ao, const std::vector<int64\_t>& shape,

const std::vector<std::string>& dim\_names,

std::shared\_ptr<SparseCSRMatrix>\* out) {

return NdarraysToSparseCSXMatrix<SparseCSRIndex>(pool, data\_ao, indptr\_ao, indices\_ao,

shape, dim\_names, out);

}

Status NdarraysToSparseCSCMatrix(MemoryPool\* pool, PyObject\* data\_ao, PyObject\* indptr\_ao,

PyObject\* indices\_ao, const std::vector<int64\_t>& shape,

const std::vector<std::string>& dim\_names,

std::shared\_ptr<SparseCSCMatrix>\* out) {

return NdarraysToSparseCSXMatrix<SparseCSCIndex>(pool, data\_ao, indptr\_ao, indices\_ao,

shape, dim\_names, out);

}

Status TensorToSparseCOOTensor(const std::shared\_ptr<Tensor>& tensor,

std::shared\_ptr<SparseCOOTensor>\* out) {

return SparseCOOTensor::Make(\*tensor).Value(out);

}

Status TensorToSparseCSRMatrix(const std::shared\_ptr<Tensor>& tensor,

std::shared\_ptr<SparseCSRMatrix>\* out) {

return SparseCSRMatrix::Make(\*tensor).Value(out);

}

Status TensorToSparseCSCMatrix(const std::shared\_ptr<Tensor>& tensor,

std::shared\_ptr<SparseCSCMatrix>\* out) {

return SparseCSCMatrix::Make(\*tensor).Value(out);

}

Status TensorToSparseCSFTensor(const std::shared\_ptr<Tensor>& tensor,

std::shared\_ptr<SparseCSFTensor>\* out) {

return SparseCSFTensor::Make(\*tensor).Value(out);

}

} // namespace py

} // namespace arrow