// 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/serialize.h"

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

#include <cstdint>

#include <limits>

#include <memory>

#include <sstream>

#include <string>

#include <vector>

#include <numpy/arrayobject.h>

#include <numpy/arrayscalars.h>

#include "arrow/array.h"

#include "arrow/array/builder\_binary.h"

#include "arrow/array/builder\_nested.h"

#include "arrow/array/builder\_primitive.h"

#include "arrow/array/builder\_union.h"

#include "arrow/io/interfaces.h"

#include "arrow/io/memory.h"

#include "arrow/ipc/util.h"

#include "arrow/ipc/writer.h"

#include "arrow/record\_batch.h"

#include "arrow/result.h"

#include "arrow/tensor.h"

#include "arrow/util/logging.h"

#include "arrow/python/common.h"

#include "arrow/python/datetime.h"

#include "arrow/python/helpers.h"

#include "arrow/python/iterators.h"

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

#include "arrow/python/platform.h"

#include "arrow/python/pyarrow.h"

constexpr int32\_t kMaxRecursionDepth = 100;

namespace arrow {

using internal::checked\_cast;

namespace py {

class SequenceBuilder;

class DictBuilder;

Status Append(PyObject\* context, PyObject\* elem, SequenceBuilder\* builder,

int32\_t recursion\_depth, SerializedPyObject\* blobs\_out);

// A Sequence is a heterogeneous collections of elements. It can contain

// scalar Python types, lists, tuples, dictionaries, tensors and sparse tensors.

class SequenceBuilder {

public:

explicit SequenceBuilder(MemoryPool\* pool = default\_memory\_pool())

: pool\_(pool),

types\_(::arrow::int8(), pool),

offsets\_(::arrow::int32(), pool),

type\_map\_(PythonType::NUM\_PYTHON\_TYPES, -1) {

auto null\_builder = std::make\_shared<NullBuilder>(pool);

auto initial\_ty = dense\_union({field("0", null())});

builder\_.reset(new DenseUnionBuilder(pool, {null\_builder}, initial\_ty));

}

// Appending a none to the sequence

Status AppendNone() { return builder\_->AppendNull(); }

template <typename BuilderType, typename MakeBuilderFn>

Status CreateAndUpdate(std::shared\_ptr<BuilderType>\* child\_builder, int8\_t tag,

MakeBuilderFn make\_builder) {

if (!\*child\_builder) {

child\_builder->reset(make\_builder());

std::ostringstream convert;

convert.imbue(std::locale::classic());

convert << static\_cast<int>(tag);

type\_map\_[tag] = builder\_->AppendChild(\*child\_builder, convert.str());

}

return builder\_->Append(type\_map\_[tag]);

}

template <typename BuilderType, typename T>

Status AppendPrimitive(std::shared\_ptr<BuilderType>\* child\_builder, const T val,

int8\_t tag) {

RETURN\_NOT\_OK(

CreateAndUpdate(child\_builder, tag, [this]() { return new BuilderType(pool\_); }));

return (\*child\_builder)->Append(val);

}

// Appending a boolean to the sequence

Status AppendBool(const bool data) {

return AppendPrimitive(&bools\_, data, PythonType::BOOL);

}

// Appending an int64\_t to the sequence

Status AppendInt64(const int64\_t data) {

return AppendPrimitive(&ints\_, data, PythonType::INT);

}

// Append a list of bytes to the sequence

Status AppendBytes(const uint8\_t\* data, int32\_t length) {

RETURN\_NOT\_OK(CreateAndUpdate(&bytes\_, PythonType::BYTES,

[this]() { return new BinaryBuilder(pool\_); }));

return bytes\_->Append(data, length);

}

// Appending a string to the sequence

Status AppendString(const char\* data, int32\_t length) {

RETURN\_NOT\_OK(CreateAndUpdate(&strings\_, PythonType::STRING,

[this]() { return new StringBuilder(pool\_); }));

return strings\_->Append(data, length);

}

// Appending a half\_float to the sequence

Status AppendHalfFloat(const npy\_half data) {

return AppendPrimitive(&half\_floats\_, data, PythonType::HALF\_FLOAT);

}

// Appending a float to the sequence

Status AppendFloat(const float data) {

return AppendPrimitive(&floats\_, data, PythonType::FLOAT);

}

// Appending a double to the sequence

Status AppendDouble(const double data) {

return AppendPrimitive(&doubles\_, data, PythonType::DOUBLE);

}

// Appending a Date64 timestamp to the sequence

Status AppendDate64(const int64\_t timestamp) {

return AppendPrimitive(&date64s\_, timestamp, PythonType::DATE64);

}

// Appending a tensor to the sequence

//

// \param tensor\_index Index of the tensor in the object.

Status AppendTensor(const int32\_t tensor\_index) {

RETURN\_NOT\_OK(CreateAndUpdate(&tensor\_indices\_, PythonType::TENSOR,

[this]() { return new Int32Builder(pool\_); }));

return tensor\_indices\_->Append(tensor\_index);

}

// Appending a sparse coo tensor to the sequence

//

// \param sparse\_coo\_tensor\_index Index of the sparse coo tensor in the object.

Status AppendSparseCOOTensor(const int32\_t sparse\_coo\_tensor\_index) {

RETURN\_NOT\_OK(CreateAndUpdate(&sparse\_coo\_tensor\_indices\_,

PythonType::SPARSECOOTENSOR,

[this]() { return new Int32Builder(pool\_); }));

return sparse\_coo\_tensor\_indices\_->Append(sparse\_coo\_tensor\_index);

}

// Appending a sparse csr matrix to the sequence

//

// \param sparse\_csr\_matrix\_index Index of the sparse csr matrix in the object.

Status AppendSparseCSRMatrix(const int32\_t sparse\_csr\_matrix\_index) {

RETURN\_NOT\_OK(CreateAndUpdate(&sparse\_csr\_matrix\_indices\_,

PythonType::SPARSECSRMATRIX,

[this]() { return new Int32Builder(pool\_); }));

return sparse\_csr\_matrix\_indices\_->Append(sparse\_csr\_matrix\_index);

}

// Appending a sparse csc matrix to the sequence

//

// \param sparse\_csc\_matrix\_index Index of the sparse csc matrix in the object.

Status AppendSparseCSCMatrix(const int32\_t sparse\_csc\_matrix\_index) {

RETURN\_NOT\_OK(CreateAndUpdate(&sparse\_csc\_matrix\_indices\_,

PythonType::SPARSECSCMATRIX,

[this]() { return new Int32Builder(pool\_); }));

return sparse\_csc\_matrix\_indices\_->Append(sparse\_csc\_matrix\_index);

}

// Appending a sparse csf tensor to the sequence

//

// \param sparse\_csf\_tensor\_index Index of the sparse csf tensor in the object.

Status AppendSparseCSFTensor(const int32\_t sparse\_csf\_tensor\_index) {

RETURN\_NOT\_OK(CreateAndUpdate(&sparse\_csf\_tensor\_indices\_,

PythonType::SPARSECSFTENSOR,

[this]() { return new Int32Builder(pool\_); }));

return sparse\_csf\_tensor\_indices\_->Append(sparse\_csf\_tensor\_index);

}

// Appending a numpy ndarray to the sequence

//

// \param tensor\_index Index of the tensor in the object.

Status AppendNdarray(const int32\_t ndarray\_index) {

RETURN\_NOT\_OK(CreateAndUpdate(&ndarray\_indices\_, PythonType::NDARRAY,

[this]() { return new Int32Builder(pool\_); }));

return ndarray\_indices\_->Append(ndarray\_index);

}

// Appending a buffer to the sequence

//

// \param buffer\_index Index of the buffer in the object.

Status AppendBuffer(const int32\_t buffer\_index) {

RETURN\_NOT\_OK(CreateAndUpdate(&buffer\_indices\_, PythonType::BUFFER,

[this]() { return new Int32Builder(pool\_); }));

return buffer\_indices\_->Append(buffer\_index);

}

Status AppendSequence(PyObject\* context, PyObject\* sequence, int8\_t tag,

std::shared\_ptr<ListBuilder>& target\_sequence,

std::unique\_ptr<SequenceBuilder>& values, int32\_t recursion\_depth,

SerializedPyObject\* blobs\_out) {

if (recursion\_depth >= kMaxRecursionDepth) {

return Status::NotImplemented(

"This object exceeds the maximum recursion depth. It may contain itself "

"recursively.");

}

RETURN\_NOT\_OK(CreateAndUpdate(&target\_sequence, tag, [this, &values]() {

values.reset(new SequenceBuilder(pool\_));

return new ListBuilder(pool\_, values->builder());

}));

RETURN\_NOT\_OK(target\_sequence->Append());

return internal::VisitIterable(

sequence, [&](PyObject\* obj, bool\* keep\_going /\* unused \*/) {

return Append(context, obj, values.get(), recursion\_depth, blobs\_out);

});

}

Status AppendList(PyObject\* context, PyObject\* list, int32\_t recursion\_depth,

SerializedPyObject\* blobs\_out) {

return AppendSequence(context, list, PythonType::LIST, lists\_, list\_values\_,

recursion\_depth + 1, blobs\_out);

}

Status AppendTuple(PyObject\* context, PyObject\* tuple, int32\_t recursion\_depth,

SerializedPyObject\* blobs\_out) {

return AppendSequence(context, tuple, PythonType::TUPLE, tuples\_, tuple\_values\_,

recursion\_depth + 1, blobs\_out);

}

Status AppendSet(PyObject\* context, PyObject\* set, int32\_t recursion\_depth,

SerializedPyObject\* blobs\_out) {

return AppendSequence(context, set, PythonType::SET, sets\_, set\_values\_,

recursion\_depth + 1, blobs\_out);

}

Status AppendDict(PyObject\* context, PyObject\* dict, int32\_t recursion\_depth,

SerializedPyObject\* blobs\_out);

// Finish building the sequence and return the result.

// Input arrays may be nullptr

Status Finish(std::shared\_ptr<Array>\* out) { return builder\_->Finish(out); }

std::shared\_ptr<DenseUnionBuilder> builder() { return builder\_; }

private:

MemoryPool\* pool\_;

Int8Builder types\_;

Int32Builder offsets\_;

/// Mapping from PythonType to child index

std::vector<int8\_t> type\_map\_;

std::shared\_ptr<BooleanBuilder> bools\_;

std::shared\_ptr<Int64Builder> ints\_;

std::shared\_ptr<BinaryBuilder> bytes\_;

std::shared\_ptr<StringBuilder> strings\_;

std::shared\_ptr<HalfFloatBuilder> half\_floats\_;

std::shared\_ptr<FloatBuilder> floats\_;

std::shared\_ptr<DoubleBuilder> doubles\_;

std::shared\_ptr<Date64Builder> date64s\_;

std::unique\_ptr<SequenceBuilder> list\_values\_;

std::shared\_ptr<ListBuilder> lists\_;

std::unique\_ptr<DictBuilder> dict\_values\_;

std::shared\_ptr<ListBuilder> dicts\_;

std::unique\_ptr<SequenceBuilder> tuple\_values\_;

std::shared\_ptr<ListBuilder> tuples\_;

std::unique\_ptr<SequenceBuilder> set\_values\_;

std::shared\_ptr<ListBuilder> sets\_;

std::shared\_ptr<Int32Builder> tensor\_indices\_;

std::shared\_ptr<Int32Builder> sparse\_coo\_tensor\_indices\_;

std::shared\_ptr<Int32Builder> sparse\_csr\_matrix\_indices\_;

std::shared\_ptr<Int32Builder> sparse\_csc\_matrix\_indices\_;

std::shared\_ptr<Int32Builder> sparse\_csf\_tensor\_indices\_;

std::shared\_ptr<Int32Builder> ndarray\_indices\_;

std::shared\_ptr<Int32Builder> buffer\_indices\_;

std::shared\_ptr<DenseUnionBuilder> builder\_;

};

// Constructing dictionaries of key/value pairs. Sequences of

// keys and values are built separately using a pair of

// SequenceBuilders. The resulting Arrow representation

// can be obtained via the Finish method.

class DictBuilder {

public:

explicit DictBuilder(MemoryPool\* pool = nullptr) : keys\_(pool), vals\_(pool) {

builder\_.reset(new StructBuilder(struct\_({field("keys", dense\_union(FieldVector{})),

field("vals", dense\_union(FieldVector{}))}),

pool, {keys\_.builder(), vals\_.builder()}));

}

// Builder for the keys of the dictionary

SequenceBuilder& keys() { return keys\_; }

// Builder for the values of the dictionary

SequenceBuilder& vals() { return vals\_; }

// Construct an Arrow StructArray representing the dictionary.

// Contains a field "keys" for the keys and "vals" for the values.

Status Finish(std::shared\_ptr<Array>\* out) { return builder\_->Finish(out); }

std::shared\_ptr<StructBuilder> builder() { return builder\_; }

private:

SequenceBuilder keys\_;

SequenceBuilder vals\_;

std::shared\_ptr<StructBuilder> builder\_;

};

Status SequenceBuilder::AppendDict(PyObject\* context, PyObject\* dict,

int32\_t recursion\_depth,

SerializedPyObject\* blobs\_out) {

if (recursion\_depth >= kMaxRecursionDepth) {

return Status::NotImplemented(

"This object exceeds the maximum recursion depth. It may contain itself "

"recursively.");

}

RETURN\_NOT\_OK(CreateAndUpdate(&dicts\_, PythonType::DICT, [this]() {

dict\_values\_.reset(new DictBuilder(pool\_));

return new ListBuilder(pool\_, dict\_values\_->builder());

}));

RETURN\_NOT\_OK(dicts\_->Append());

PyObject\* key;

PyObject\* value;

Py\_ssize\_t pos = 0;

while (PyDict\_Next(dict, &pos, &key, &value)) {

RETURN\_NOT\_OK(dict\_values\_->builder()->Append());

RETURN\_NOT\_OK(

Append(context, key, &dict\_values\_->keys(), recursion\_depth + 1, blobs\_out));

RETURN\_NOT\_OK(

Append(context, value, &dict\_values\_->vals(), recursion\_depth + 1, blobs\_out));

}

// This block is used to decrement the reference counts of the results

// returned by the serialization callback, which is called in AppendArray,

// in DeserializeDict and in Append

static PyObject\* py\_type = PyUnicode\_FromString("\_pytype\_");

if (PyDict\_Contains(dict, py\_type)) {

// If the dictionary contains the key "\_pytype\_", then the user has to

// have registered a callback.

if (context == Py\_None) {

return Status::Invalid("No serialization callback set");

}

Py\_XDECREF(dict);

}

return Status::OK();

}

Status CallCustomCallback(PyObject\* context, PyObject\* method\_name, PyObject\* elem,

PyObject\*\* result) {

if (context == Py\_None) {

\*result = NULL;

return Status::SerializationError("error while calling callback on ",

internal::PyObject\_StdStringRepr(elem),

": handler not registered");

} else {

\*result = PyObject\_CallMethodObjArgs(context, method\_name, elem, NULL);

return CheckPyError();

}

}

Status CallSerializeCallback(PyObject\* context, PyObject\* value,

PyObject\*\* serialized\_object) {

OwnedRef method\_name(PyUnicode\_FromString("\_serialize\_callback"));

RETURN\_NOT\_OK(CallCustomCallback(context, method\_name.obj(), value, serialized\_object));

if (!PyDict\_Check(\*serialized\_object)) {

return Status::TypeError("serialization callback must return a valid dictionary");

}

return Status::OK();

}

Status CallDeserializeCallback(PyObject\* context, PyObject\* value,

PyObject\*\* deserialized\_object) {

OwnedRef method\_name(PyUnicode\_FromString("\_deserialize\_callback"));

return CallCustomCallback(context, method\_name.obj(), value, deserialized\_object);

}

Status AppendArray(PyObject\* context, PyArrayObject\* array, SequenceBuilder\* builder,

int32\_t recursion\_depth, SerializedPyObject\* blobs\_out);

template <typename NumpyScalarObject>

Status AppendIntegerScalar(PyObject\* obj, SequenceBuilder\* builder) {

int64\_t value = reinterpret\_cast<NumpyScalarObject\*>(obj)->obval;

return builder->AppendInt64(value);

}

// Append a potentially 64-bit wide unsigned Numpy scalar.

// Must check for overflow as we reinterpret it as signed int64.

template <typename NumpyScalarObject>

Status AppendLargeUnsignedScalar(PyObject\* obj, SequenceBuilder\* builder) {

constexpr uint64\_t max\_value = std::numeric\_limits<int64\_t>::max();

uint64\_t value = reinterpret\_cast<NumpyScalarObject\*>(obj)->obval;

if (value > max\_value) {

return Status::Invalid("cannot serialize Numpy uint64 scalar >= 2\*\*63");

}

return builder->AppendInt64(static\_cast<int64\_t>(value));

}

Status AppendScalar(PyObject\* obj, SequenceBuilder\* builder) {

if (PyArray\_IsScalar(obj, Bool)) {

return builder->AppendBool(reinterpret\_cast<PyBoolScalarObject\*>(obj)->obval != 0);

} else if (PyArray\_IsScalar(obj, Half)) {

return builder->AppendHalfFloat(reinterpret\_cast<PyHalfScalarObject\*>(obj)->obval);

} else if (PyArray\_IsScalar(obj, Float)) {

return builder->AppendFloat(reinterpret\_cast<PyFloatScalarObject\*>(obj)->obval);

} else if (PyArray\_IsScalar(obj, Double)) {

return builder->AppendDouble(reinterpret\_cast<PyDoubleScalarObject\*>(obj)->obval);

}

if (PyArray\_IsScalar(obj, Byte)) {

return AppendIntegerScalar<PyByteScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, Short)) {

return AppendIntegerScalar<PyShortScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, Int)) {

return AppendIntegerScalar<PyIntScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, Long)) {

return AppendIntegerScalar<PyLongScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, LongLong)) {

return AppendIntegerScalar<PyLongLongScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, Int64)) {

return AppendIntegerScalar<PyInt64ScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, UByte)) {

return AppendIntegerScalar<PyUByteScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, UShort)) {

return AppendIntegerScalar<PyUShortScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, UInt)) {

return AppendIntegerScalar<PyUIntScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, ULong)) {

return AppendLargeUnsignedScalar<PyULongScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, ULongLong)) {

return AppendLargeUnsignedScalar<PyULongLongScalarObject>(obj, builder);

} else if (PyArray\_IsScalar(obj, UInt64)) {

return AppendLargeUnsignedScalar<PyUInt64ScalarObject>(obj, builder);

}

return Status::NotImplemented("Numpy scalar type not recognized");

}

Status Append(PyObject\* context, PyObject\* elem, SequenceBuilder\* builder,

int32\_t recursion\_depth, SerializedPyObject\* blobs\_out) {

// The bool case must precede the int case (PyInt\_Check passes for bools)

if (PyBool\_Check(elem)) {

RETURN\_NOT\_OK(builder->AppendBool(elem == Py\_True));

} else if (PyArray\_DescrFromScalar(elem)->type\_num == NPY\_HALF) {

npy\_half halffloat = reinterpret\_cast<PyHalfScalarObject\*>(elem)->obval;

RETURN\_NOT\_OK(builder->AppendHalfFloat(halffloat));

} else if (PyFloat\_Check(elem)) {

RETURN\_NOT\_OK(builder->AppendDouble(PyFloat\_AS\_DOUBLE(elem)));

} else if (PyLong\_Check(elem)) {

int overflow = 0;

int64\_t data = PyLong\_AsLongLongAndOverflow(elem, &overflow);

if (!overflow) {

RETURN\_NOT\_OK(builder->AppendInt64(data));

} else {

// Attempt to serialize the object using the custom callback.

PyObject\* serialized\_object;

// The reference count of serialized\_object will be decremented in SerializeDict

RETURN\_NOT\_OK(CallSerializeCallback(context, elem, &serialized\_object));

RETURN\_NOT\_OK(

builder->AppendDict(context, serialized\_object, recursion\_depth, blobs\_out));

}

} else if (PyBytes\_Check(elem)) {

auto data = reinterpret\_cast<uint8\_t\*>(PyBytes\_AS\_STRING(elem));

int32\_t size = -1;

RETURN\_NOT\_OK(internal::CastSize(PyBytes\_GET\_SIZE(elem), &size));

RETURN\_NOT\_OK(builder->AppendBytes(data, size));

} else if (PyUnicode\_Check(elem)) {

ARROW\_ASSIGN\_OR\_RAISE(auto view, PyBytesView::FromUnicode(elem));

int32\_t size = -1;

RETURN\_NOT\_OK(internal::CastSize(view.size, &size));

RETURN\_NOT\_OK(builder->AppendString(view.bytes, size));

} else if (PyList\_CheckExact(elem)) {

RETURN\_NOT\_OK(builder->AppendList(context, elem, recursion\_depth, blobs\_out));

} else if (PyDict\_CheckExact(elem)) {

RETURN\_NOT\_OK(builder->AppendDict(context, elem, recursion\_depth, blobs\_out));

} else if (PyTuple\_CheckExact(elem)) {

RETURN\_NOT\_OK(builder->AppendTuple(context, elem, recursion\_depth, blobs\_out));

} else if (PySet\_Check(elem)) {

RETURN\_NOT\_OK(builder->AppendSet(context, elem, recursion\_depth, blobs\_out));

} else if (PyArray\_IsScalar(elem, Generic)) {

RETURN\_NOT\_OK(AppendScalar(elem, builder));

} else if (PyArray\_CheckExact(elem)) {

RETURN\_NOT\_OK(AppendArray(context, reinterpret\_cast<PyArrayObject\*>(elem), builder,

recursion\_depth, blobs\_out));

} else if (elem == Py\_None) {

RETURN\_NOT\_OK(builder->AppendNone());

} else if (PyDateTime\_Check(elem)) {

PyDateTime\_DateTime\* datetime = reinterpret\_cast<PyDateTime\_DateTime\*>(elem);

RETURN\_NOT\_OK(builder->AppendDate64(internal::PyDateTime\_to\_us(datetime)));

} else if (is\_buffer(elem)) {

RETURN\_NOT\_OK(builder->AppendBuffer(static\_cast<int32\_t>(blobs\_out->buffers.size())));

ARROW\_ASSIGN\_OR\_RAISE(auto buffer, unwrap\_buffer(elem));

blobs\_out->buffers.push\_back(buffer);

} else if (is\_tensor(elem)) {

RETURN\_NOT\_OK(builder->AppendTensor(static\_cast<int32\_t>(blobs\_out->tensors.size())));

ARROW\_ASSIGN\_OR\_RAISE(auto tensor, unwrap\_tensor(elem));

blobs\_out->tensors.push\_back(tensor);

} else if (is\_sparse\_coo\_tensor(elem)) {

RETURN\_NOT\_OK(builder->AppendSparseCOOTensor(

static\_cast<int32\_t>(blobs\_out->sparse\_tensors.size())));

ARROW\_ASSIGN\_OR\_RAISE(auto tensor, unwrap\_sparse\_coo\_tensor(elem));

blobs\_out->sparse\_tensors.push\_back(tensor);

} else if (is\_sparse\_csr\_matrix(elem)) {

RETURN\_NOT\_OK(builder->AppendSparseCSRMatrix(

static\_cast<int32\_t>(blobs\_out->sparse\_tensors.size())));

ARROW\_ASSIGN\_OR\_RAISE(auto matrix, unwrap\_sparse\_csr\_matrix(elem));

blobs\_out->sparse\_tensors.push\_back(matrix);

} else if (is\_sparse\_csc\_matrix(elem)) {

RETURN\_NOT\_OK(builder->AppendSparseCSCMatrix(

static\_cast<int32\_t>(blobs\_out->sparse\_tensors.size())));

ARROW\_ASSIGN\_OR\_RAISE(auto matrix, unwrap\_sparse\_csc\_matrix(elem));

blobs\_out->sparse\_tensors.push\_back(matrix);

} else if (is\_sparse\_csf\_tensor(elem)) {

RETURN\_NOT\_OK(builder->AppendSparseCSFTensor(

static\_cast<int32\_t>(blobs\_out->sparse\_tensors.size())));

ARROW\_ASSIGN\_OR\_RAISE(auto tensor, unwrap\_sparse\_csf\_tensor(elem));

blobs\_out->sparse\_tensors.push\_back(tensor);

} else {

// Attempt to serialize the object using the custom callback.

PyObject\* serialized\_object;

// The reference count of serialized\_object will be decremented in SerializeDict

RETURN\_NOT\_OK(CallSerializeCallback(context, elem, &serialized\_object));

RETURN\_NOT\_OK(

builder->AppendDict(context, serialized\_object, recursion\_depth, blobs\_out));

}

return Status::OK();

}

Status AppendArray(PyObject\* context, PyArrayObject\* array, SequenceBuilder\* builder,

int32\_t recursion\_depth, SerializedPyObject\* blobs\_out) {

int dtype = PyArray\_TYPE(array);

switch (dtype) {

case NPY\_UINT8:

case NPY\_INT8:

case NPY\_UINT16:

case NPY\_INT16:

case NPY\_UINT32:

case NPY\_INT32:

case NPY\_UINT64:

case NPY\_INT64:

case NPY\_HALF:

case NPY\_FLOAT:

case NPY\_DOUBLE: {

RETURN\_NOT\_OK(

builder->AppendNdarray(static\_cast<int32\_t>(blobs\_out->ndarrays.size())));

std::shared\_ptr<Tensor> tensor;

RETURN\_NOT\_OK(NdarrayToTensor(default\_memory\_pool(),

reinterpret\_cast<PyObject\*>(array), {}, &tensor));

blobs\_out->ndarrays.push\_back(tensor);

} break;

default: {

PyObject\* serialized\_object;

// The reference count of serialized\_object will be decremented in SerializeDict

RETURN\_NOT\_OK(CallSerializeCallback(context, reinterpret\_cast<PyObject\*>(array),

&serialized\_object));

RETURN\_NOT\_OK(builder->AppendDict(context, serialized\_object, recursion\_depth + 1,

blobs\_out));

}

}

return Status::OK();

}

std::shared\_ptr<RecordBatch> MakeBatch(std::shared\_ptr<Array> data) {

auto field = std::make\_shared<Field>("list", data->type());

auto schema = ::arrow::schema({field});

return RecordBatch::Make(schema, data->length(), {data});

}

Status SerializeObject(PyObject\* context, PyObject\* sequence, SerializedPyObject\* out) {

PyAcquireGIL lock;

SequenceBuilder builder;

RETURN\_NOT\_OK(internal::VisitIterable(

sequence, [&](PyObject\* obj, bool\* keep\_going /\* unused \*/) {

return Append(context, obj, &builder, 0, out);

}));

std::shared\_ptr<Array> array;

RETURN\_NOT\_OK(builder.Finish(&array));

out->batch = MakeBatch(array);

return Status::OK();

}

Status SerializeNdarray(std::shared\_ptr<Tensor> tensor, SerializedPyObject\* out) {

std::shared\_ptr<Array> array;

SequenceBuilder builder;

RETURN\_NOT\_OK(builder.AppendNdarray(static\_cast<int32\_t>(out->ndarrays.size())));

out->ndarrays.push\_back(tensor);

RETURN\_NOT\_OK(builder.Finish(&array));

out->batch = MakeBatch(array);

return Status::OK();

}

Status WriteNdarrayHeader(std::shared\_ptr<DataType> dtype,

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

io::OutputStream\* dst) {

auto empty\_tensor = std::make\_shared<Tensor>(

dtype, std::make\_shared<Buffer>(nullptr, tensor\_num\_bytes), shape);

SerializedPyObject serialized\_tensor;

RETURN\_NOT\_OK(SerializeNdarray(empty\_tensor, &serialized\_tensor));

return serialized\_tensor.WriteTo(dst);

}

SerializedPyObject::SerializedPyObject()

: ipc\_options(ipc::IpcWriteOptions::Defaults()) {}

Status SerializedPyObject::WriteTo(io::OutputStream\* dst) {

int32\_t num\_tensors = static\_cast<int32\_t>(this->tensors.size());

int32\_t num\_sparse\_tensors = static\_cast<int32\_t>(this->sparse\_tensors.size());

int32\_t num\_ndarrays = static\_cast<int32\_t>(this->ndarrays.size());

int32\_t num\_buffers = static\_cast<int32\_t>(this->buffers.size());

RETURN\_NOT\_OK(

dst->Write(reinterpret\_cast<const uint8\_t\*>(&num\_tensors), sizeof(int32\_t)));

RETURN\_NOT\_OK(

dst->Write(reinterpret\_cast<const uint8\_t\*>(&num\_sparse\_tensors), sizeof(int32\_t)));

RETURN\_NOT\_OK(

dst->Write(reinterpret\_cast<const uint8\_t\*>(&num\_ndarrays), sizeof(int32\_t)));

RETURN\_NOT\_OK(

dst->Write(reinterpret\_cast<const uint8\_t\*>(&num\_buffers), sizeof(int32\_t)));

// Align stream to 8-byte offset

RETURN\_NOT\_OK(ipc::AlignStream(dst, ipc::kArrowIpcAlignment));

RETURN\_NOT\_OK(ipc::WriteRecordBatchStream({this->batch}, this->ipc\_options, dst));

// Align stream to 64-byte offset so tensor bodies are 64-byte aligned

RETURN\_NOT\_OK(ipc::AlignStream(dst, ipc::kTensorAlignment));

int32\_t metadata\_length;

int64\_t body\_length;

for (const auto& tensor : this->tensors) {

RETURN\_NOT\_OK(ipc::WriteTensor(\*tensor, dst, &metadata\_length, &body\_length));

RETURN\_NOT\_OK(ipc::AlignStream(dst, ipc::kTensorAlignment));

}

for (const auto& sparse\_tensor : this->sparse\_tensors) {

RETURN\_NOT\_OK(

ipc::WriteSparseTensor(\*sparse\_tensor, dst, &metadata\_length, &body\_length));

RETURN\_NOT\_OK(ipc::AlignStream(dst, ipc::kTensorAlignment));

}

for (const auto& tensor : this->ndarrays) {

RETURN\_NOT\_OK(ipc::WriteTensor(\*tensor, dst, &metadata\_length, &body\_length));

RETURN\_NOT\_OK(ipc::AlignStream(dst, ipc::kTensorAlignment));

}

for (const auto& buffer : this->buffers) {

int64\_t size = buffer->size();

RETURN\_NOT\_OK(dst->Write(reinterpret\_cast<const uint8\_t\*>(&size), sizeof(int64\_t)));

RETURN\_NOT\_OK(dst->Write(buffer->data(), size));

}

return Status::OK();

}

namespace {

Status CountSparseTensors(

const std::vector<std::shared\_ptr<SparseTensor>>& sparse\_tensors, PyObject\*\* out) {

OwnedRef num\_sparse\_tensors(PyDict\_New());

size\_t num\_coo = 0;

size\_t num\_csr = 0;

size\_t num\_csc = 0;

size\_t num\_csf = 0;

size\_t ndim\_csf = 0;

for (const auto& sparse\_tensor : sparse\_tensors) {

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

case SparseTensorFormat::COO:

++num\_coo;

break;

case SparseTensorFormat::CSR:

++num\_csr;

break;

case SparseTensorFormat::CSC:

++num\_csc;

break;

case SparseTensorFormat::CSF:

++num\_csf;

ndim\_csf += sparse\_tensor->ndim();

break;

}

}

PyDict\_SetItemString(num\_sparse\_tensors.obj(), "coo", PyLong\_FromSize\_t(num\_coo));

PyDict\_SetItemString(num\_sparse\_tensors.obj(), "csr", PyLong\_FromSize\_t(num\_csr));

PyDict\_SetItemString(num\_sparse\_tensors.obj(), "csc", PyLong\_FromSize\_t(num\_csc));

PyDict\_SetItemString(num\_sparse\_tensors.obj(), "csf", PyLong\_FromSize\_t(num\_csf));

PyDict\_SetItemString(num\_sparse\_tensors.obj(), "ndim\_csf", PyLong\_FromSize\_t(ndim\_csf));

RETURN\_IF\_PYERROR();

\*out = num\_sparse\_tensors.detach();

return Status::OK();

}

} // namespace

Status SerializedPyObject::GetComponents(MemoryPool\* memory\_pool, PyObject\*\* out) {

PyAcquireGIL py\_gil;

OwnedRef result(PyDict\_New());

PyObject\* buffers = PyList\_New(0);

PyObject\* num\_sparse\_tensors = nullptr;

// TODO(wesm): Not sure how pedantic we need to be about checking the return

// values of these functions. There are other places where we do not check

// PyDict\_SetItem/SetItemString return value, but these failures would be

// quite esoteric

PyDict\_SetItemString(result.obj(), "num\_tensors",

PyLong\_FromSize\_t(this->tensors.size()));

RETURN\_NOT\_OK(CountSparseTensors(this->sparse\_tensors, &num\_sparse\_tensors));

PyDict\_SetItemString(result.obj(), "num\_sparse\_tensors", num\_sparse\_tensors);

PyDict\_SetItemString(result.obj(), "ndim\_csf", num\_sparse\_tensors);

PyDict\_SetItemString(result.obj(), "num\_ndarrays",

PyLong\_FromSize\_t(this->ndarrays.size()));

PyDict\_SetItemString(result.obj(), "num\_buffers",

PyLong\_FromSize\_t(this->buffers.size()));

PyDict\_SetItemString(result.obj(), "data", buffers);

RETURN\_IF\_PYERROR();

Py\_DECREF(buffers);

auto PushBuffer = [&buffers](const std::shared\_ptr<Buffer>& buffer) {

PyObject\* wrapped\_buffer = wrap\_buffer(buffer);

RETURN\_IF\_PYERROR();

if (PyList\_Append(buffers, wrapped\_buffer) < 0) {

Py\_DECREF(wrapped\_buffer);

RETURN\_IF\_PYERROR();

}

Py\_DECREF(wrapped\_buffer);

return Status::OK();

};

constexpr int64\_t kInitialCapacity = 1024;

// Write the record batch describing the object structure

py\_gil.release();

ARROW\_ASSIGN\_OR\_RAISE(auto stream,

io::BufferOutputStream::Create(kInitialCapacity, memory\_pool));

RETURN\_NOT\_OK(

ipc::WriteRecordBatchStream({this->batch}, this->ipc\_options, stream.get()));

ARROW\_ASSIGN\_OR\_RAISE(auto buffer, stream->Finish());

py\_gil.acquire();

RETURN\_NOT\_OK(PushBuffer(buffer));

// For each tensor, get a metadata buffer and a buffer for the body

for (const auto& tensor : this->tensors) {

ARROW\_ASSIGN\_OR\_RAISE(std::unique\_ptr<ipc::Message> message,

ipc::GetTensorMessage(\*tensor, memory\_pool));

RETURN\_NOT\_OK(PushBuffer(message->metadata()));

RETURN\_NOT\_OK(PushBuffer(message->body()));

}

// For each sparse tensor, get a metadata buffer and buffers containing index and data

for (const auto& sparse\_tensor : this->sparse\_tensors) {

ipc::IpcPayload payload;

RETURN\_NOT\_OK(ipc::GetSparseTensorPayload(\*sparse\_tensor, memory\_pool, &payload));

RETURN\_NOT\_OK(PushBuffer(payload.metadata));

for (const auto& body : payload.body\_buffers) {

RETURN\_NOT\_OK(PushBuffer(body));

}

}

// For each ndarray, get a metadata buffer and a buffer for the body

for (const auto& ndarray : this->ndarrays) {

ARROW\_ASSIGN\_OR\_RAISE(std::unique\_ptr<ipc::Message> message,

ipc::GetTensorMessage(\*ndarray, memory\_pool));

RETURN\_NOT\_OK(PushBuffer(message->metadata()));

RETURN\_NOT\_OK(PushBuffer(message->body()));

}

for (const auto& buf : this->buffers) {

RETURN\_NOT\_OK(PushBuffer(buf));

}

\*out = result.detach();

return Status::OK();

}

} // namespace py

} // namespace arrow