// 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/deserialize.h"

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

#include <cstdint>

#include <memory>

#include <string>

#include <utility>

#include <vector>

#include <numpy/arrayobject.h>

#include <numpy/arrayscalars.h>

#include "arrow/array.h"

#include "arrow/io/interfaces.h"

#include "arrow/io/memory.h"

#include "arrow/ipc/options.h"

#include "arrow/ipc/reader.h"

#include "arrow/ipc/util.h"

#include "arrow/ipc/writer.h"

#include "arrow/table.h"

#include "arrow/util/checked\_cast.h"

#include "arrow/util/logging.h"

#include "arrow/util/value\_parsing.h"

#include "arrow/python/common.h"

#include "arrow/python/datetime.h"

#include "arrow/python/helpers.h"

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

#include "arrow/python/pyarrow.h"

#include "arrow/python/serialize.h"

#include "arrow/python/vendored/pythoncapi\_compat.h"

namespace arrow {

using internal::checked\_cast;

using internal::ParseValue;

namespace py {

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

PyObject\*\* deserialized\_object);

Status DeserializeTuple(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out);

Status DeserializeList(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out);

Status DeserializeSet(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out);

Status DeserializeDict(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out) {

const auto& data = checked\_cast<const StructArray&>(array);

OwnedRef keys, vals;

OwnedRef result(PyDict\_New());

RETURN\_IF\_PYERROR();

DCHECK\_EQ(2, data.num\_fields());

RETURN\_NOT\_OK(DeserializeList(context, \*data.field(0), start\_idx, stop\_idx, base, blobs,

keys.ref()));

RETURN\_NOT\_OK(DeserializeList(context, \*data.field(1), start\_idx, stop\_idx, base, blobs,

vals.ref()));

for (int64\_t i = start\_idx; i < stop\_idx; ++i) {

// PyDict\_SetItem behaves differently from PyList\_SetItem and PyTuple\_SetItem.

// The latter two steal references whereas PyDict\_SetItem does not. So we need

// to make sure the reference count is decremented by letting the OwnedRef

// go out of scope at the end.

PyObject\* key = PyList\_GetItemRef(keys.obj(), i - start\_idx);

RETURN\_IF\_PYERROR();

OwnedRef keyref(key);

PyObject\* val = PyList\_GetItemRef(vals.obj(), i - start\_idx);

RETURN\_IF\_PYERROR();

OwnedRef valref(val);

int ret = PyDict\_SetItem(result.obj(), key, val);

if (ret != 0) {

return ConvertPyError();

}

}

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

if (PyDict\_Contains(result.obj(), py\_type)) {

RETURN\_NOT\_OK(CallDeserializeCallback(context, result.obj(), out));

} else {

\*out = result.detach();

}

return Status::OK();

}

Status DeserializeArray(int32\_t index, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out) {

RETURN\_NOT\_OK(py::TensorToNdarray(blobs.ndarrays[index], base, out));

// Mark the array as immutable

OwnedRef flags(PyObject\_GetAttrString(\*out, "flags"));

if (flags.obj() == NULL) {

return ConvertPyError();

}

if (PyObject\_SetAttrString(flags.obj(), "writeable", Py\_False) < 0) {

return ConvertPyError();

}

return Status::OK();

}

Status GetValue(PyObject\* context, const Array& arr, int64\_t index, int8\_t type,

PyObject\* base, const SerializedPyObject& blobs, PyObject\*\* result) {

switch (type) {

case PythonType::NONE:

Py\_INCREF(Py\_None);

\*result = Py\_None;

return Status::OK();

case PythonType::BOOL:

\*result = PyBool\_FromLong(checked\_cast<const BooleanArray&>(arr).Value(index));

return Status::OK();

case PythonType::PY2INT:

case PythonType::INT: {

\*result = PyLong\_FromSsize\_t(checked\_cast<const Int64Array&>(arr).Value(index));

return Status::OK();

}

case PythonType::BYTES: {

auto view = checked\_cast<const BinaryArray&>(arr).GetView(index);

\*result = PyBytes\_FromStringAndSize(view.data(), view.length());

return CheckPyError();

}

case PythonType::STRING: {

auto view = checked\_cast<const StringArray&>(arr).GetView(index);

\*result = PyUnicode\_FromStringAndSize(view.data(), view.length());

return CheckPyError();

}

case PythonType::HALF\_FLOAT: {

\*result = PyHalf\_FromHalf(checked\_cast<const HalfFloatArray&>(arr).Value(index));

RETURN\_IF\_PYERROR();

return Status::OK();

}

case PythonType::FLOAT:

\*result = PyFloat\_FromDouble(checked\_cast<const FloatArray&>(arr).Value(index));

return Status::OK();

case PythonType::DOUBLE:

\*result = PyFloat\_FromDouble(checked\_cast<const DoubleArray&>(arr).Value(index));

return Status::OK();

case PythonType::DATE64: {

RETURN\_NOT\_OK(internal::PyDateTime\_from\_int(

checked\_cast<const Date64Array&>(arr).Value(index), TimeUnit::MICRO, result));

RETURN\_IF\_PYERROR();

return Status::OK();

}

case PythonType::LIST: {

const auto& l = checked\_cast<const ListArray&>(arr);

return DeserializeList(context, \*l.values(), l.value\_offset(index),

l.value\_offset(index + 1), base, blobs, result);

}

case PythonType::DICT: {

const auto& l = checked\_cast<const ListArray&>(arr);

return DeserializeDict(context, \*l.values(), l.value\_offset(index),

l.value\_offset(index + 1), base, blobs, result);

}

case PythonType::TUPLE: {

const auto& l = checked\_cast<const ListArray&>(arr);

return DeserializeTuple(context, \*l.values(), l.value\_offset(index),

l.value\_offset(index + 1), base, blobs, result);

}

case PythonType::SET: {

const auto& l = checked\_cast<const ListArray&>(arr);

return DeserializeSet(context, \*l.values(), l.value\_offset(index),

l.value\_offset(index + 1), base, blobs, result);

}

case PythonType::TENSOR: {

int32\_t ref = checked\_cast<const Int32Array&>(arr).Value(index);

\*result = wrap\_tensor(blobs.tensors[ref]);

return Status::OK();

}

case PythonType::SPARSECOOTENSOR: {

int32\_t ref = checked\_cast<const Int32Array&>(arr).Value(index);

const std::shared\_ptr<SparseCOOTensor>& sparse\_coo\_tensor =

arrow::internal::checked\_pointer\_cast<SparseCOOTensor>(

blobs.sparse\_tensors[ref]);

\*result = wrap\_sparse\_coo\_tensor(sparse\_coo\_tensor);

return Status::OK();

}

case PythonType::SPARSECSRMATRIX: {

int32\_t ref = checked\_cast<const Int32Array&>(arr).Value(index);

const std::shared\_ptr<SparseCSRMatrix>& sparse\_csr\_matrix =

arrow::internal::checked\_pointer\_cast<SparseCSRMatrix>(

blobs.sparse\_tensors[ref]);

\*result = wrap\_sparse\_csr\_matrix(sparse\_csr\_matrix);

return Status::OK();

}

case PythonType::SPARSECSCMATRIX: {

int32\_t ref = checked\_cast<const Int32Array&>(arr).Value(index);

const std::shared\_ptr<SparseCSCMatrix>& sparse\_csc\_matrix =

arrow::internal::checked\_pointer\_cast<SparseCSCMatrix>(

blobs.sparse\_tensors[ref]);

\*result = wrap\_sparse\_csc\_matrix(sparse\_csc\_matrix);

return Status::OK();

}

case PythonType::SPARSECSFTENSOR: {

int32\_t ref = checked\_cast<const Int32Array&>(arr).Value(index);

const std::shared\_ptr<SparseCSFTensor>& sparse\_csf\_tensor =

arrow::internal::checked\_pointer\_cast<SparseCSFTensor>(

blobs.sparse\_tensors[ref]);

\*result = wrap\_sparse\_csf\_tensor(sparse\_csf\_tensor);

return Status::OK();

}

case PythonType::NDARRAY: {

int32\_t ref = checked\_cast<const Int32Array&>(arr).Value(index);

return DeserializeArray(ref, base, blobs, result);

}

case PythonType::BUFFER: {

int32\_t ref = checked\_cast<const Int32Array&>(arr).Value(index);

\*result = wrap\_buffer(blobs.buffers[ref]);

return Status::OK();

}

default: {

ARROW\_CHECK(false) << "union tag " << type << "' not recognized";

}

}

return Status::OK();

}

Status GetPythonTypes(const UnionArray& data, std::vector<int8\_t>\* result) {

ARROW\_CHECK(result != nullptr);

auto type = data.type();

for (int i = 0; i < type->num\_fields(); ++i) {

int8\_t tag = 0;

const std::string& data = type->field(i)->name();

if (!ParseValue<Int8Type>(data.c\_str(), data.size(), &tag)) {

return Status::SerializationError("Cannot convert string: \"",

type->field(i)->name(), "\" to int8\_t");

}

result->push\_back(tag);

}

return Status::OK();

}

template <typename CreateSequenceFn, typename SetItemFn>

Status DeserializeSequence(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base,

const SerializedPyObject& blobs,

CreateSequenceFn&& create\_sequence, SetItemFn&& set\_item,

PyObject\*\* out) {

const auto& data = checked\_cast<const DenseUnionArray&>(array);

OwnedRef result(create\_sequence(stop\_idx - start\_idx));

RETURN\_IF\_PYERROR();

const int8\_t\* type\_codes = data.raw\_type\_codes();

const int32\_t\* value\_offsets = data.raw\_value\_offsets();

std::vector<int8\_t> python\_types;

RETURN\_NOT\_OK(GetPythonTypes(data, &python\_types));

for (int64\_t i = start\_idx; i < stop\_idx; ++i) {

const int64\_t offset = value\_offsets[i];

const uint8\_t type = type\_codes[i];

PyObject\* value;

RETURN\_NOT\_OK(GetValue(context, \*data.field(type), offset, python\_types[type], base,

blobs, &value));

RETURN\_NOT\_OK(set\_item(result.obj(), i - start\_idx, value));

}

\*out = result.detach();

return Status::OK();

}

Status DeserializeList(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out) {

return DeserializeSequence(

context, array, start\_idx, stop\_idx, base, blobs,

[](int64\_t size) { return PyList\_New(size); },

[](PyObject\* seq, int64\_t index, PyObject\* item) {

PyList\_SET\_ITEM(seq, index, item);

return Status::OK();

},

out);

}

Status DeserializeTuple(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out) {

return DeserializeSequence(

context, array, start\_idx, stop\_idx, base, blobs,

[](int64\_t size) { return PyTuple\_New(size); },

[](PyObject\* seq, int64\_t index, PyObject\* item) {

PyTuple\_SET\_ITEM(seq, index, item);

return Status::OK();

},

out);

}

Status DeserializeSet(PyObject\* context, const Array& array, int64\_t start\_idx,

int64\_t stop\_idx, PyObject\* base, const SerializedPyObject& blobs,

PyObject\*\* out) {

return DeserializeSequence(

context, array, start\_idx, stop\_idx, base, blobs,

[](int64\_t size) { return PySet\_New(nullptr); },

[](PyObject\* seq, int64\_t index, PyObject\* item) {

int err = PySet\_Add(seq, item);

Py\_DECREF(item);

if (err < 0) {

RETURN\_IF\_PYERROR();

}

return Status::OK();

},

out);

}

Status ReadSerializedObject(io::RandomAccessFile\* src, SerializedPyObject\* out) {

int32\_t num\_tensors;

int32\_t num\_sparse\_tensors;

int32\_t num\_ndarrays;

int32\_t num\_buffers;

// Read number of tensors

RETURN\_NOT\_OK(src->Read(sizeof(int32\_t), reinterpret\_cast<uint8\_t\*>(&num\_tensors)));

RETURN\_NOT\_OK(

src->Read(sizeof(int32\_t), reinterpret\_cast<uint8\_t\*>(&num\_sparse\_tensors)));

RETURN\_NOT\_OK(src->Read(sizeof(int32\_t), reinterpret\_cast<uint8\_t\*>(&num\_ndarrays)));

RETURN\_NOT\_OK(src->Read(sizeof(int32\_t), reinterpret\_cast<uint8\_t\*>(&num\_buffers)));

// Align stream to 8-byte offset

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

std::shared\_ptr<RecordBatchReader> reader;

ARROW\_ASSIGN\_OR\_RAISE(reader, ipc::RecordBatchStreamReader::Open(src));

RETURN\_NOT\_OK(reader->ReadNext(&out->batch));

/// Skip EOS marker

RETURN\_NOT\_OK(src->Advance(4));

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

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

for (int i = 0; i < num\_tensors; ++i) {

std::shared\_ptr<Tensor> tensor;

ARROW\_ASSIGN\_OR\_RAISE(tensor, ipc::ReadTensor(src));

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

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

}

for (int i = 0; i < num\_sparse\_tensors; ++i) {

std::shared\_ptr<SparseTensor> sparse\_tensor;

ARROW\_ASSIGN\_OR\_RAISE(sparse\_tensor, ipc::ReadSparseTensor(src));

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

out->sparse\_tensors.push\_back(sparse\_tensor);

}

for (int i = 0; i < num\_ndarrays; ++i) {

std::shared\_ptr<Tensor> ndarray;

ARROW\_ASSIGN\_OR\_RAISE(ndarray, ipc::ReadTensor(src));

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

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

}

ARROW\_ASSIGN\_OR\_RAISE(int64\_t offset, src->Tell());

for (int i = 0; i < num\_buffers; ++i) {

int64\_t size;

RETURN\_NOT\_OK(src->ReadAt(offset, sizeof(int64\_t), &size));

offset += sizeof(int64\_t);

ARROW\_ASSIGN\_OR\_RAISE(auto buffer, src->ReadAt(offset, size));

out->buffers.push\_back(buffer);

offset += size;

}

return Status::OK();

}

Status DeserializeObject(PyObject\* context, const SerializedPyObject& obj, PyObject\* base,

PyObject\*\* out) {

PyAcquireGIL lock;

return DeserializeList(context, \*obj.batch->column(0), 0, obj.batch->num\_rows(), base,

obj, out);

}

Status GetSerializedFromComponents(int num\_tensors,

const SparseTensorCounts& num\_sparse\_tensors,

int num\_ndarrays, int num\_buffers, PyObject\* data,

SerializedPyObject\* out) {

PyAcquireGIL gil;

const Py\_ssize\_t data\_length = PyList\_Size(data);

RETURN\_IF\_PYERROR();

const Py\_ssize\_t expected\_data\_length = 1 + num\_tensors \* 2 +

num\_sparse\_tensors.num\_total\_buffers() +

num\_ndarrays \* 2 + num\_buffers;

if (data\_length != expected\_data\_length) {

return Status::Invalid("Invalid number of buffers in data");

}

auto GetBuffer = [&data](Py\_ssize\_t index, std::shared\_ptr<Buffer>\* out) {

ARROW\_CHECK\_LE(index, PyList\_Size(data));

PyObject\* py\_buf = PyList\_GetItemRef(data, index);

RETURN\_IF\_PYERROR();

OwnedRef py\_buf\_ref(py\_buf);

return unwrap\_buffer(py\_buf).Value(out);

};

Py\_ssize\_t buffer\_index = 0;

// Read the union batch describing object structure

{

std::shared\_ptr<Buffer> data\_buffer;

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &data\_buffer));

gil.release();

io::BufferReader buf\_reader(data\_buffer);

std::shared\_ptr<RecordBatchReader> reader;

ARROW\_ASSIGN\_OR\_RAISE(reader, ipc::RecordBatchStreamReader::Open(&buf\_reader));

RETURN\_NOT\_OK(reader->ReadNext(&out->batch));

gil.acquire();

}

// Zero-copy reconstruct tensors

for (int i = 0; i < num\_tensors; ++i) {

std::shared\_ptr<Buffer> metadata;

std::shared\_ptr<Buffer> body;

std::shared\_ptr<Tensor> tensor;

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &metadata));

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &body));

ipc::Message message(metadata, body);

ARROW\_ASSIGN\_OR\_RAISE(tensor, ipc::ReadTensor(message));

out->tensors.emplace\_back(std::move(tensor));

}

// Zero-copy reconstruct sparse tensors

for (int i = 0, n = num\_sparse\_tensors.num\_total\_tensors(); i < n; ++i) {

ipc::IpcPayload payload;

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &payload.metadata));

ARROW\_ASSIGN\_OR\_RAISE(

size\_t num\_bodies,

ipc::internal::ReadSparseTensorBodyBufferCount(\*payload.metadata));

payload.body\_buffers.reserve(num\_bodies);

for (size\_t i = 0; i < num\_bodies; ++i) {

std::shared\_ptr<Buffer> body;

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &body));

payload.body\_buffers.emplace\_back(body);

}

std::shared\_ptr<SparseTensor> sparse\_tensor;

ARROW\_ASSIGN\_OR\_RAISE(sparse\_tensor, ipc::internal::ReadSparseTensorPayload(payload));

out->sparse\_tensors.emplace\_back(std::move(sparse\_tensor));

}

// Zero-copy reconstruct tensors for numpy ndarrays

for (int i = 0; i < num\_ndarrays; ++i) {

std::shared\_ptr<Buffer> metadata;

std::shared\_ptr<Buffer> body;

std::shared\_ptr<Tensor> tensor;

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &metadata));

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &body));

ipc::Message message(metadata, body);

ARROW\_ASSIGN\_OR\_RAISE(tensor, ipc::ReadTensor(message));

out->ndarrays.emplace\_back(std::move(tensor));

}

// Unwrap and append buffers

for (int i = 0; i < num\_buffers; ++i) {

std::shared\_ptr<Buffer> buffer;

RETURN\_NOT\_OK(GetBuffer(buffer\_index++, &buffer));

out->buffers.emplace\_back(std::move(buffer));

}

return Status::OK();

}

Status DeserializeNdarray(const SerializedPyObject& object,

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

if (object.ndarrays.size() != 1) {

return Status::Invalid("Object is not an Ndarray");

}

\*out = object.ndarrays[0];

return Status::OK();

}

Status NdarrayFromBuffer(std::shared\_ptr<Buffer> src, std::shared\_ptr<Tensor>\* out) {

io::BufferReader reader(src);

SerializedPyObject object;

RETURN\_NOT\_OK(ReadSerializedObject(&reader, &object));

return DeserializeNdarray(object, out);

}

} // namespace py

} // namespace arrow