// 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.

// Functions for pandas conversion via NumPy

#include "arrow/python/numpy\_to\_arrow.h"

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

#include <algorithm>

#include <cmath>

#include <cstdint>

#include <cstring>

#include <limits>

#include <memory>

#include <string>

#include <utility>

#include <vector>

#include "arrow/array.h"

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

#include "arrow/status.h"

#include "arrow/table.h"

#include "arrow/type\_fwd.h"

#include "arrow/type\_traits.h"

#include "arrow/util/bit\_util.h"

#include "arrow/util/bitmap\_generate.h"

#include "arrow/util/bitmap\_ops.h"

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

#include "arrow/util/endian.h"

#include "arrow/util/logging.h"

#include "arrow/util/macros.h"

#include "arrow/util/string.h"

#include "arrow/util/utf8.h"

#include "arrow/visit\_type\_inline.h"

#include "arrow/compute/api\_scalar.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/numpy\_internal.h"

#include "arrow/python/python\_to\_arrow.h"

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

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

namespace arrow {

using internal::checked\_cast;

using internal::CopyBitmap;

using internal::GenerateBitsUnrolled;

namespace py {

using internal::NumPyTypeSize;

// ----------------------------------------------------------------------

// Conversion utilities

namespace {

Status AllocateNullBitmap(MemoryPool\* pool, int64\_t length,

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

int64\_t null\_bytes = bit\_util::BytesForBits(length);

ARROW\_ASSIGN\_OR\_RAISE(auto null\_bitmap, AllocateResizableBuffer(null\_bytes, pool));

// Padding zeroed by AllocateResizableBuffer

memset(null\_bitmap->mutable\_data(), 0, static\_cast<size\_t>(null\_bytes));

\*out = std::move(null\_bitmap);

return Status::OK();

}

// ----------------------------------------------------------------------

// Conversion from NumPy-in-Pandas to Arrow null bitmap

template <int TYPE>

inline int64\_t ValuesToBitmap(PyArrayObject\* arr, uint8\_t\* bitmap) {

typedef internal::npy\_traits<TYPE> traits;

typedef typename traits::value\_type T;

int64\_t null\_count = 0;

Ndarray1DIndexer<T> values(arr);

for (int i = 0; i < values.size(); ++i) {

if (traits::isnull(values[i])) {

++null\_count;

} else {

bit\_util::SetBit(bitmap, i);

}

}

return null\_count;

}

class NumPyNullsConverter {

public:

/// Convert the given array's null values to a null bitmap.

/// The null bitmap is only allocated if null values are ever possible.

static Status Convert(MemoryPool\* pool, PyArrayObject\* arr, bool from\_pandas,

std::shared\_ptr<ResizableBuffer>\* out\_null\_bitmap\_,

int64\_t\* out\_null\_count) {

NumPyNullsConverter converter(pool, arr, from\_pandas);

RETURN\_NOT\_OK(VisitNumpyArrayInline(arr, &converter));

\*out\_null\_bitmap\_ = converter.null\_bitmap\_;

\*out\_null\_count = converter.null\_count\_;

return Status::OK();

}

template <int TYPE>

Status Visit(PyArrayObject\* arr) {

typedef internal::npy\_traits<TYPE> traits;

const bool null\_sentinels\_possible =

// Always treat Numpy's NaT as null

TYPE == NPY\_DATETIME || TYPE == NPY\_TIMEDELTA ||

// Observing pandas's null sentinels

(from\_pandas\_ && traits::supports\_nulls);

if (null\_sentinels\_possible) {

RETURN\_NOT\_OK(AllocateNullBitmap(pool\_, PyArray\_SIZE(arr), &null\_bitmap\_));

null\_count\_ = ValuesToBitmap<TYPE>(arr, null\_bitmap\_->mutable\_data());

}

return Status::OK();

}

protected:

NumPyNullsConverter(MemoryPool\* pool, PyArrayObject\* arr, bool from\_pandas)

: pool\_(pool),

arr\_(arr),

from\_pandas\_(from\_pandas),

null\_bitmap\_data\_(nullptr),

null\_count\_(0) {}

MemoryPool\* pool\_;

PyArrayObject\* arr\_;

bool from\_pandas\_;

std::shared\_ptr<ResizableBuffer> null\_bitmap\_;

uint8\_t\* null\_bitmap\_data\_;

int64\_t null\_count\_;

};

// Returns null count

int64\_t MaskToBitmap(PyArrayObject\* mask, int64\_t length, uint8\_t\* bitmap) {

int64\_t null\_count = 0;

if (!PyArray\_Check(mask)) return -1;

Ndarray1DIndexer<uint8\_t> mask\_values(mask);

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

if (mask\_values[i]) {

++null\_count;

bit\_util::ClearBit(bitmap, i);

} else {

bit\_util::SetBit(bitmap, i);

}

}

return null\_count;

}

} // namespace

// ----------------------------------------------------------------------

// Conversion from NumPy arrays (possibly originating from pandas) to Arrow

// format. Does not handle NPY\_OBJECT dtype arrays; use ConvertPySequence for

// that

class NumPyConverter {

public:

NumPyConverter(MemoryPool\* pool, PyObject\* arr, PyObject\* mo,

const std::shared\_ptr<DataType>& type, bool from\_pandas,

const compute::CastOptions& cast\_options = compute::CastOptions())

: pool\_(pool),

type\_(type),

arr\_(reinterpret\_cast<PyArrayObject\*>(arr)),

dtype\_(PyArray\_DESCR(arr\_)),

mask\_(nullptr),

from\_pandas\_(from\_pandas),

cast\_options\_(cast\_options),

null\_bitmap\_data\_(nullptr),

null\_count\_(0) {

if (mo != nullptr && mo != Py\_None) {

mask\_ = reinterpret\_cast<PyArrayObject\*>(mo);

}

length\_ = static\_cast<int64\_t>(PyArray\_SIZE(arr\_));

itemsize\_ = static\_cast<int64\_t>(PyArray\_ITEMSIZE(arr\_));

stride\_ = static\_cast<int64\_t>(PyArray\_STRIDES(arr\_)[0]);

}

bool is\_strided() const { return itemsize\_ != stride\_; }

Status Convert();

const ArrayVector& result() const { return out\_arrays\_; }

template <typename T>

enable\_if\_primitive\_ctype<T, Status> Visit(const T& type) {

return VisitNative<T>();

}

Status Visit(const HalfFloatType& type) { return VisitNative<UInt16Type>(); }

Status Visit(const Date32Type& type) { return VisitNative<Date32Type>(); }

Status Visit(const Date64Type& type) { return VisitNative<Date64Type>(); }

Status Visit(const TimestampType& type) { return VisitNative<TimestampType>(); }

Status Visit(const Time32Type& type) { return VisitNative<Int32Type>(); }

Status Visit(const Time64Type& type) { return VisitNative<Int64Type>(); }

Status Visit(const DurationType& type) { return VisitNative<DurationType>(); }

Status Visit(const NullType& type) { return TypeNotImplemented(type.ToString()); }

// NumPy ascii string arrays

Status Visit(const BinaryType& type);

// NumPy unicode arrays

Status Visit(const StringType& type);

Status Visit(const StructType& type);

Status Visit(const FixedSizeBinaryType& type);

// Default case

Status Visit(const DataType& type) { return TypeNotImplemented(type.ToString()); }

protected:

Status InitNullBitmap() {

RETURN\_NOT\_OK(AllocateNullBitmap(pool\_, length\_, &null\_bitmap\_));

null\_bitmap\_data\_ = null\_bitmap\_->mutable\_data();

return Status::OK();

}

// Called before ConvertData to ensure Numpy input buffer is in expected

// Arrow layout

template <typename ArrowType>

Status PrepareInputData(std::shared\_ptr<Buffer>\* data);

// ----------------------------------------------------------------------

// Traditional visitor conversion for non-object arrays

template <typename ArrowType>

Status ConvertData(std::shared\_ptr<Buffer>\* data);

template <typename T>

Status PushBuilderResult(T\* builder) {

std::shared\_ptr<Array> out;

RETURN\_NOT\_OK(builder->Finish(&out));

out\_arrays\_.emplace\_back(out);

return Status::OK();

}

Status PushArray(const std::shared\_ptr<ArrayData>& data) {

out\_arrays\_.emplace\_back(MakeArray(data));

return Status::OK();

}

template <typename ArrowType>

Status VisitNative() {

if (mask\_ != nullptr) {

RETURN\_NOT\_OK(InitNullBitmap());

null\_count\_ = MaskToBitmap(mask\_, length\_, null\_bitmap\_data\_);

if (null\_count\_ == -1) return Status::Invalid("Invalid mask type");

} else {

RETURN\_NOT\_OK(NumPyNullsConverter::Convert(pool\_, arr\_, from\_pandas\_, &null\_bitmap\_,

&null\_count\_));

}

std::shared\_ptr<Buffer> data;

RETURN\_NOT\_OK(ConvertData<ArrowType>(&data));

auto arr\_data = ArrayData::Make(type\_, length\_, {null\_bitmap\_, data}, null\_count\_, 0);

return PushArray(arr\_data);

}

Status TypeNotImplemented(std::string type\_name) {

return Status::NotImplemented("NumPyConverter doesn't implement <", type\_name,

"> conversion. ");

}

MemoryPool\* pool\_;

std::shared\_ptr<DataType> type\_;

PyArrayObject\* arr\_;

PyArray\_Descr\* dtype\_;

PyArrayObject\* mask\_;

int64\_t length\_;

int64\_t stride\_;

int64\_t itemsize\_;

bool from\_pandas\_;

compute::CastOptions cast\_options\_;

// Used in visitor pattern

ArrayVector out\_arrays\_;

std::shared\_ptr<ResizableBuffer> null\_bitmap\_;

uint8\_t\* null\_bitmap\_data\_;

int64\_t null\_count\_;

};

Status NumPyConverter::Convert() {

if (PyArray\_NDIM(arr\_) != 1) {

return Status::Invalid("only handle 1-dimensional arrays");

}

if (dtype\_->type\_num == NPY\_OBJECT) {

// If an object array, convert it like a normal Python sequence

PyConversionOptions py\_options;

py\_options.type = type\_;

py\_options.from\_pandas = from\_pandas\_;

ARROW\_ASSIGN\_OR\_RAISE(

auto chunked\_array,

ConvertPySequence(reinterpret\_cast<PyObject\*>(arr\_),

reinterpret\_cast<PyObject\*>(mask\_), py\_options, pool\_));

out\_arrays\_ = chunked\_array->chunks();

return Status::OK();

}

if (type\_ == nullptr) {

return Status::Invalid("Must pass data type for non-object arrays");

}

// Visit the type to perform conversion

return VisitTypeInline(\*type\_, this);

}

namespace {

Status CastBuffer(const std::shared\_ptr<DataType>& in\_type,

const std::shared\_ptr<Buffer>& input, const int64\_t length,

const std::shared\_ptr<Buffer>& valid\_bitmap, const int64\_t null\_count,

const std::shared\_ptr<DataType>& out\_type,

const compute::CastOptions& cast\_options, MemoryPool\* pool,

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

// Must cast

auto tmp\_data = ArrayData::Make(in\_type, length, {valid\_bitmap, input}, null\_count);

compute::ExecContext context(pool);

ARROW\_ASSIGN\_OR\_RAISE(

std::shared\_ptr<Array> casted\_array,

compute::Cast(\*MakeArray(tmp\_data), out\_type, cast\_options, &context));

\*out = casted\_array->data()->buffers[1];

return Status::OK();

}

template <typename FromType, typename ToType>

Status StaticCastBuffer(const Buffer& input, const int64\_t length, MemoryPool\* pool,

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

ARROW\_ASSIGN\_OR\_RAISE(auto result, AllocateBuffer(sizeof(ToType) \* length, pool));

auto in\_values = reinterpret\_cast<const FromType\*>(input.data());

auto out\_values = reinterpret\_cast<ToType\*>(result->mutable\_data());

for (int64\_t i = 0; i < length; ++i) {

\*out\_values++ = static\_cast<ToType>(\*in\_values++);

}

\*out = std::move(result);

return Status::OK();

}

template <typename T>

void CopyStridedBytewise(int8\_t\* input\_data, int64\_t length, int64\_t stride,

T\* output\_data) {

// Passing input\_data as non-const is a concession to PyObject\*

for (int64\_t i = 0; i < length; ++i) {

memcpy(output\_data + i, input\_data, sizeof(T));

input\_data += stride;

}

}

template <typename T>

void CopyStridedNatural(T\* input\_data, int64\_t length, int64\_t stride, T\* output\_data) {

// Passing input\_data as non-const is a concession to PyObject\*

int64\_t j = 0;

for (int64\_t i = 0; i < length; ++i) {

output\_data[i] = input\_data[j];

j += stride;

}

}

class NumPyStridedConverter {

public:

static Status Convert(PyArrayObject\* arr, int64\_t length, MemoryPool\* pool,

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

NumPyStridedConverter converter(arr, length, pool);

RETURN\_NOT\_OK(VisitNumpyArrayInline(arr, &converter));

\*out = converter.buffer\_;

return Status::OK();

}

template <int TYPE>

Status Visit(PyArrayObject\* arr) {

using traits = internal::npy\_traits<TYPE>;

using T = typename traits::value\_type;

ARROW\_ASSIGN\_OR\_RAISE(buffer\_, AllocateBuffer(sizeof(T) \* length\_, pool\_));

const int64\_t stride = PyArray\_STRIDES(arr)[0];

// ARROW-16013: convert sizeof(T) to signed int64 first, otherwise dividing by it

// would do an unsigned division. This cannot be caught by tests without ubsan, since

// common signed overflow behavior and the fact that the sizeof(T) is currently always

// a power of two here cause CopyStridedNatural to still produce correct results

const int64\_t element\_size = sizeof(T);

if (stride % element\_size == 0) {

const int64\_t stride\_elements = stride / element\_size;

CopyStridedNatural(reinterpret\_cast<T\*>(PyArray\_DATA(arr)), length\_,

stride\_elements, reinterpret\_cast<T\*>(buffer\_->mutable\_data()));

} else {

CopyStridedBytewise(reinterpret\_cast<int8\_t\*>(PyArray\_DATA(arr)), length\_, stride,

reinterpret\_cast<T\*>(buffer\_->mutable\_data()));

}

return Status::OK();

}

protected:

NumPyStridedConverter(PyArrayObject\* arr, int64\_t length, MemoryPool\* pool)

: arr\_(arr), length\_(length), pool\_(pool), buffer\_(nullptr) {}

PyArrayObject\* arr\_;

int64\_t length\_;

MemoryPool\* pool\_;

std::shared\_ptr<Buffer> buffer\_;

};

} // namespace

template <typename ArrowType>

inline Status NumPyConverter::PrepareInputData(std::shared\_ptr<Buffer>\* data) {

if (PyArray\_ISBYTESWAPPED(arr\_)) {

// TODO

return Status::NotImplemented("Byte-swapped arrays not supported");

}

if (dtype\_->type\_num == NPY\_BOOL) {

int64\_t nbytes = bit\_util::BytesForBits(length\_);

ARROW\_ASSIGN\_OR\_RAISE(auto buffer, AllocateBuffer(nbytes, pool\_));

Ndarray1DIndexer<uint8\_t> values(arr\_);

int64\_t i = 0;

const auto generate = [&values, &i]() -> bool { return values[i++] > 0; };

GenerateBitsUnrolled(buffer->mutable\_data(), 0, length\_, generate);

\*data = std::move(buffer);

} else if (is\_strided()) {

RETURN\_NOT\_OK(NumPyStridedConverter::Convert(arr\_, length\_, pool\_, data));

} else {

// Can zero-copy

\*data = std::make\_shared<NumPyBuffer>(reinterpret\_cast<PyObject\*>(arr\_));

}

return Status::OK();

}

template <typename ArrowType>

inline Status NumPyConverter::ConvertData(std::shared\_ptr<Buffer>\* data) {

RETURN\_NOT\_OK(PrepareInputData<ArrowType>(data));

ARROW\_ASSIGN\_OR\_RAISE(auto input\_type, NumPyDtypeToArrow(dtype\_));

if (!input\_type->Equals(\*type\_)) {

RETURN\_NOT\_OK(CastBuffer(input\_type, \*data, length\_, null\_bitmap\_, null\_count\_, type\_,

cast\_options\_, pool\_, data));

}

return Status::OK();

}

template <>

inline Status NumPyConverter::ConvertData<Date32Type>(std::shared\_ptr<Buffer>\* data) {

std::shared\_ptr<DataType> input\_type;

RETURN\_NOT\_OK(PrepareInputData<Date32Type>(data));

auto date\_dtype =

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

if (dtype\_->type\_num == NPY\_DATETIME) {

// If we have inbound datetime64[D] data, this needs to be downcasted

// separately here from int64\_t to int32\_t, because this data is not

// supported in compute::Cast

if (date\_dtype->meta.base == NPY\_FR\_D) {

// TODO(wesm): How pedantic do we really want to be about checking for int32

// overflow here?

Status s = StaticCastBuffer<int64\_t, int32\_t>(\*\*data, length\_, pool\_, data);

RETURN\_NOT\_OK(s);

} else {

ARROW\_ASSIGN\_OR\_RAISE(input\_type, NumPyDtypeToArrow(dtype\_));

if (!input\_type->Equals(\*type\_)) {

// The null bitmap was already computed in VisitNative()

RETURN\_NOT\_OK(CastBuffer(input\_type, \*data, length\_, null\_bitmap\_, null\_count\_,

type\_, cast\_options\_, pool\_, data));

}

}

} else {

ARROW\_ASSIGN\_OR\_RAISE(input\_type, NumPyDtypeToArrow(dtype\_));

if (!input\_type->Equals(\*type\_)) {

RETURN\_NOT\_OK(CastBuffer(input\_type, \*data, length\_, null\_bitmap\_, null\_count\_,

type\_, cast\_options\_, pool\_, data));

}

}

return Status::OK();

}

template <>

inline Status NumPyConverter::ConvertData<Date64Type>(std::shared\_ptr<Buffer>\* data) {

constexpr int64\_t kMillisecondsInDay = 86400000;

std::shared\_ptr<DataType> input\_type;

RETURN\_NOT\_OK(PrepareInputData<Date64Type>(data));

auto date\_dtype =

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

if (dtype\_->type\_num == NPY\_DATETIME) {

// If we have inbound datetime64[D] data, this needs to be downcasted

// separately here from int64\_t to int32\_t, because this data is not

// supported in compute::Cast

if (date\_dtype->meta.base == NPY\_FR\_D) {

ARROW\_ASSIGN\_OR\_RAISE(auto result,

AllocateBuffer(sizeof(int64\_t) \* length\_, pool\_));

auto in\_values = reinterpret\_cast<const int64\_t\*>((\*data)->data());

auto out\_values = reinterpret\_cast<int64\_t\*>(result->mutable\_data());

for (int64\_t i = 0; i < length\_; ++i) {

\*out\_values++ = kMillisecondsInDay \* (\*in\_values++);

}

\*data = std::move(result);

} else {

ARROW\_ASSIGN\_OR\_RAISE(input\_type, NumPyDtypeToArrow(dtype\_));

if (!input\_type->Equals(\*type\_)) {

// The null bitmap was already computed in VisitNative()

RETURN\_NOT\_OK(CastBuffer(input\_type, \*data, length\_, null\_bitmap\_, null\_count\_,

type\_, cast\_options\_, pool\_, data));

}

}

} else {

ARROW\_ASSIGN\_OR\_RAISE(input\_type, NumPyDtypeToArrow(dtype\_));

if (!input\_type->Equals(\*type\_)) {

RETURN\_NOT\_OK(CastBuffer(input\_type, \*data, length\_, null\_bitmap\_, null\_count\_,

type\_, cast\_options\_, pool\_, data));

}

}

return Status::OK();

}

// Create 16MB chunks for binary data

constexpr int32\_t kBinaryChunksize = 1 << 24;

Status NumPyConverter::Visit(const BinaryType& type) {

::arrow::internal::ChunkedBinaryBuilder builder(kBinaryChunksize, pool\_);

auto data = reinterpret\_cast<const uint8\_t\*>(PyArray\_DATA(arr\_));

auto AppendNotNull = [&builder, this](const uint8\_t\* data) {

// This is annoying. NumPy allows strings to have nul-terminators, so

// we must check for them here

const size\_t item\_size =

strnlen(reinterpret\_cast<const char\*>(data), static\_cast<size\_t>(itemsize\_));

return builder.Append(data, static\_cast<int32\_t>(item\_size));

};

if (mask\_ != nullptr) {

Ndarray1DIndexer<uint8\_t> mask\_values(mask\_);

for (int64\_t i = 0; i < length\_; ++i) {

if (mask\_values[i]) {

RETURN\_NOT\_OK(builder.AppendNull());

} else {

RETURN\_NOT\_OK(AppendNotNull(data));

}

data += stride\_;

}

} else {

for (int64\_t i = 0; i < length\_; ++i) {

RETURN\_NOT\_OK(AppendNotNull(data));

data += stride\_;

}

}

ArrayVector result;

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

for (auto arr : result) {

RETURN\_NOT\_OK(PushArray(arr->data()));

}

return Status::OK();

}

Status NumPyConverter::Visit(const FixedSizeBinaryType& type) {

auto byte\_width = type.byte\_width();

if (itemsize\_ != byte\_width) {

return Status::Invalid("Got bytestring of length ", itemsize\_, " (expected ",

byte\_width, ")");

}

FixedSizeBinaryBuilder builder(::arrow::fixed\_size\_binary(byte\_width), pool\_);

auto data = reinterpret\_cast<const uint8\_t\*>(PyArray\_DATA(arr\_));

if (mask\_ != nullptr) {

Ndarray1DIndexer<uint8\_t> mask\_values(mask\_);

RETURN\_NOT\_OK(builder.Reserve(length\_));

for (int64\_t i = 0; i < length\_; ++i) {

if (mask\_values[i]) {

RETURN\_NOT\_OK(builder.AppendNull());

} else {

RETURN\_NOT\_OK(builder.Append(data));

}

data += stride\_;

}

} else {

for (int64\_t i = 0; i < length\_; ++i) {

RETURN\_NOT\_OK(builder.Append(data));

data += stride\_;

}

}

std::shared\_ptr<Array> result;

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

return PushArray(result->data());

}

namespace {

// NumPy unicode is UCS4/UTF32 always

constexpr int kNumPyUnicodeSize = 4;

Status AppendUTF32(const char\* data, int64\_t itemsize, int byteorder,

::arrow::internal::ChunkedStringBuilder\* builder) {

// The binary \x00\x00\x00\x00 indicates a nul terminator in NumPy unicode,

// so we need to detect that here to truncate if necessary. Yep.

Py\_ssize\_t actual\_length = 0;

for (; actual\_length < itemsize / kNumPyUnicodeSize; ++actual\_length) {

const char\* code\_point = data + actual\_length \* kNumPyUnicodeSize;

if ((\*code\_point == '\0') && (\*(code\_point + 1) == '\0') &&

(\*(code\_point + 2) == '\0') && (\*(code\_point + 3) == '\0')) {

break;

}

}

OwnedRef unicode\_obj(PyUnicode\_DecodeUTF32(data, actual\_length \* kNumPyUnicodeSize,

nullptr, &byteorder));

RETURN\_IF\_PYERROR();

OwnedRef utf8\_obj(PyUnicode\_AsUTF8String(unicode\_obj.obj()));

if (utf8\_obj.obj() == NULL) {

PyErr\_Clear();

return Status::Invalid("failed converting UTF32 to UTF8");

}

const int32\_t length = static\_cast<int32\_t>(PyBytes\_GET\_SIZE(utf8\_obj.obj()));

return builder->Append(

reinterpret\_cast<const uint8\_t\*>(PyBytes\_AS\_STRING(utf8\_obj.obj())), length);

}

} // namespace

Status NumPyConverter::Visit(const StringType& type) {

util::InitializeUTF8();

::arrow::internal::ChunkedStringBuilder builder(kBinaryChunksize, pool\_);

auto data = reinterpret\_cast<const uint8\_t\*>(PyArray\_DATA(arr\_));

char numpy\_byteorder = dtype\_->byteorder;

// For Python C API, -1 is little-endian, 1 is big-endian

#if ARROW\_LITTLE\_ENDIAN

// Yield little-endian from both '|' (native) and '<'

int byteorder = numpy\_byteorder == '>' ? 1 : -1;

#else

// Yield big-endian from both '|' (native) and '>'

int byteorder = numpy\_byteorder == '<' ? -1 : 1;

#endif

PyAcquireGIL gil\_lock;

const bool is\_binary\_type = dtype\_->type\_num == NPY\_STRING;

const bool is\_unicode\_type = dtype\_->type\_num == NPY\_UNICODE;

if (!is\_binary\_type && !is\_unicode\_type) {

const bool is\_float\_type = dtype\_->kind == 'f';

if (from\_pandas\_ && is\_float\_type) {

// in case of from\_pandas=True, accept an all-NaN float array as input

RETURN\_NOT\_OK(NumPyNullsConverter::Convert(pool\_, arr\_, from\_pandas\_, &null\_bitmap\_,

&null\_count\_));

if (null\_count\_ == length\_) {

auto arr = std::make\_shared<NullArray>(length\_);

compute::ExecContext context(pool\_);

ARROW\_ASSIGN\_OR\_RAISE(

std::shared\_ptr<Array> out,

compute::Cast(\*arr, arrow::utf8(), cast\_options\_, &context));

out\_arrays\_.emplace\_back(out);

return Status::OK();

}

}

std::string dtype\_string;

RETURN\_NOT\_OK(internal::PyObject\_StdStringStr(reinterpret\_cast<PyObject\*>(dtype\_),

&dtype\_string));

return Status::TypeError("Expected a string or bytes dtype, got ", dtype\_string);

}

auto AppendNonNullValue = [&](const uint8\_t\* data) {

if (is\_binary\_type) {

if (ARROW\_PREDICT\_TRUE(util::ValidateUTF8(data, itemsize\_))) {

return builder.Append(data, static\_cast<int32\_t>(itemsize\_));

} else {

return Status::Invalid("Encountered non-UTF8 binary value: ",

HexEncode(data, itemsize\_));

}

} else {

// is\_unicode\_type case

return AppendUTF32(reinterpret\_cast<const char\*>(data), itemsize\_, byteorder,

&builder);

}

};

if (mask\_ != nullptr) {

Ndarray1DIndexer<uint8\_t> mask\_values(mask\_);

for (int64\_t i = 0; i < length\_; ++i) {

if (mask\_values[i]) {

RETURN\_NOT\_OK(builder.AppendNull());

} else {

RETURN\_NOT\_OK(AppendNonNullValue(data));

}

data += stride\_;

}

} else {

for (int64\_t i = 0; i < length\_; ++i) {

RETURN\_NOT\_OK(AppendNonNullValue(data));

data += stride\_;

}

}

ArrayVector result;

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

for (auto arr : result) {

RETURN\_NOT\_OK(PushArray(arr->data()));

}

return Status::OK();

}

Status NumPyConverter::Visit(const StructType& type) {

std::vector<NumPyConverter> sub\_converters;

std::vector<OwnedRefNoGIL> sub\_arrays;

{

PyAcquireGIL gil\_lock;

// Create converters for each struct type field

if (PyDataType\_FIELDS(dtype\_) == NULL || !PyDict\_Check(PyDataType\_FIELDS(dtype\_))) {

return Status::TypeError("Expected struct array");

}

for (auto field : type.fields()) {

PyObject\* tup;

PyDict\_GetItemStringRef(PyDataType\_FIELDS(dtype\_), field->name().c\_str(), &tup);

RETURN\_IF\_PYERROR();

OwnedRef tupref(tup);

if (tup == NULL) {

return Status::Invalid("Missing field '", field->name(), "' in struct array");

}

PyArray\_Descr\* sub\_dtype =

reinterpret\_cast<PyArray\_Descr\*>(PyTuple\_GET\_ITEM(tup, 0));

DCHECK(PyObject\_TypeCheck(sub\_dtype, &PyArrayDescr\_Type));

int offset = static\_cast<int>(PyLong\_AsLong(PyTuple\_GET\_ITEM(tup, 1)));

RETURN\_IF\_PYERROR();

Py\_INCREF(sub\_dtype); /\* PyArray\_GetField() steals ref \*/

PyObject\* sub\_array = PyArray\_GetField(arr\_, sub\_dtype, offset);

RETURN\_IF\_PYERROR();

sub\_arrays.emplace\_back(sub\_array);

sub\_converters.emplace\_back(pool\_, sub\_array, nullptr /\* mask \*/, field->type(),

from\_pandas\_);

}

}

std::vector<ArrayVector> groups;

int64\_t null\_count = 0;

// Compute null bitmap and store it as a Boolean Array to include it

// in the rechunking below

{

if (mask\_ != nullptr) {

RETURN\_NOT\_OK(InitNullBitmap());

null\_count = MaskToBitmap(mask\_, length\_, null\_bitmap\_data\_);

if (null\_count\_ == -1) return Status::Invalid("Invalid mask type");

}

groups.push\_back({std::make\_shared<BooleanArray>(length\_, null\_bitmap\_)});

}

// Convert child data

for (auto& converter : sub\_converters) {

RETURN\_NOT\_OK(converter.Convert());

groups.push\_back(converter.result());

}

// Ensure the different array groups are chunked consistently

groups = ::arrow::internal::RechunkArraysConsistently(groups);

// Make struct array chunks by combining groups

size\_t ngroups = groups.size();

size\_t nchunks = groups[0].size();

for (size\_t chunk = 0; chunk < nchunks; chunk++) {

// First group has the null bitmaps as Boolean Arrays

const auto& null\_data = groups[0][chunk]->data();

DCHECK\_EQ(null\_data->type->id(), Type::BOOL);

DCHECK\_EQ(null\_data->buffers.size(), 2);

const auto& null\_buffer = null\_data->buffers[1];

// Careful: the rechunked null bitmap may have a non-zero offset

// to its buffer, and it may not even start on a byte boundary

int64\_t null\_offset = null\_data->offset;

std::shared\_ptr<Buffer> fixed\_null\_buffer;

if (!null\_buffer) {

fixed\_null\_buffer = null\_buffer;

} else if (null\_offset % 8 == 0) {

fixed\_null\_buffer =

std::make\_shared<Buffer>(null\_buffer,

// byte offset

null\_offset / 8,

// byte size

bit\_util::BytesForBits(null\_data->length));

} else {

ARROW\_ASSIGN\_OR\_RAISE(

fixed\_null\_buffer,

CopyBitmap(pool\_, null\_buffer->data(), null\_offset, null\_data->length));

}

// Create struct array chunk and populate it

auto arr\_data =

ArrayData::Make(type\_, null\_data->length, null\_count ? kUnknownNullCount : 0, 0);

arr\_data->buffers.push\_back(fixed\_null\_buffer);

// Append child chunks

for (size\_t i = 1; i < ngroups; i++) {

arr\_data->child\_data.push\_back(groups[i][chunk]->data());

}

RETURN\_NOT\_OK(PushArray(arr\_data));

}

return Status::OK();

}

Status NdarrayToArrow(MemoryPool\* pool, PyObject\* ao, PyObject\* mo, bool from\_pandas,

const std::shared\_ptr<DataType>& type,

const compute::CastOptions& cast\_options,

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

if (!PyArray\_Check(ao)) {

// This code path cannot be reached by Python unit tests currently so this

// is only a sanity check.

return Status::TypeError("Input object was not a NumPy array");

}

if (PyArray\_NDIM(reinterpret\_cast<PyArrayObject\*>(ao)) != 1) {

return Status::Invalid("only handle 1-dimensional arrays");

}

NumPyConverter converter(pool, ao, mo, type, from\_pandas, cast\_options);

RETURN\_NOT\_OK(converter.Convert());

const auto& output\_arrays = converter.result();

DCHECK\_GT(output\_arrays.size(), 0);

\*out = std::make\_shared<ChunkedArray>(output\_arrays);

return Status::OK();

}

Status NdarrayToArrow(MemoryPool\* pool, PyObject\* ao, PyObject\* mo, bool from\_pandas,

const std::shared\_ptr<DataType>& type,

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

return NdarrayToArrow(pool, ao, mo, from\_pandas, type, compute::CastOptions(), out);

}

} // namespace py

} // namespace arrow