// 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/arrow\_to\_pandas.h"

#include "arrow/python/numpy\_interop.h" // IWYU pragma: expand

#include <cmath>

#include <cstdint>

#include <iostream>

#include <memory>

#include <mutex>

#include <string>

#include <string\_view>

#include <unordered\_map>

#include <utility>

#include <vector>

#include "arrow/array.h"

#include "arrow/buffer.h"

#include "arrow/datum.h"

#include "arrow/status.h"

#include "arrow/table.h"

#include "arrow/type.h"

#include "arrow/type\_traits.h"

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

#include "arrow/util/hashing.h"

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

#include "arrow/util/logging.h"

#include "arrow/util/macros.h"

#include "arrow/util/parallel.h"

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

#include "arrow/compute/api.h"

#include "arrow/python/arrow\_to\_python\_internal.h"

#include "arrow/python/common.h"

#include "arrow/python/datetime.h"

#include "arrow/python/decimal.h"

#include "arrow/python/helpers.h"

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

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

#include "arrow/python/pyarrow.h"

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

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

namespace arrow {

class MemoryPool;

using internal::checked\_cast;

using internal::CheckIndexBounds;

using internal::OptionalParallelFor;

namespace py {

namespace {

// Fix options for conversion of an inner (child) array.

PandasOptions MakeInnerOptions(PandasOptions options) {

// Make sure conversion of inner dictionary arrays always returns an array,

// not a dict {'indices': array, 'dictionary': array, 'ordered': bool}

options.decode\_dictionaries = true;

options.categorical\_columns.clear();

options.strings\_to\_categorical = false;

// In ARROW-7723, we found as a result of ARROW-3789 that second

// through microsecond resolution tz-aware timestamps were being promoted to

// use the DATETIME\_NANO\_TZ conversion path, yielding a datetime64[ns] NumPy

// array in this function. PyArray\_GETITEM returns datetime.datetime for

// units second through microsecond but PyLong for nanosecond (because

// datetime.datetime does not support nanoseconds).

// We force the object conversion to preserve the value of the timezone.

// Nanoseconds are returned as integers.

options.coerce\_temporal\_nanoseconds = false;

return options;

}

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

// PyCapsule code for setting ndarray base to reference C++ object

struct ArrayCapsule {

std::shared\_ptr<Array> array;

};

struct BufferCapsule {

std::shared\_ptr<Buffer> buffer;

};

void ArrayCapsule\_Destructor(PyObject\* capsule) {

delete reinterpret\_cast<ArrayCapsule\*>(PyCapsule\_GetPointer(capsule, "arrow::Array"));

}

void BufferCapsule\_Destructor(PyObject\* capsule) {

delete reinterpret\_cast<BufferCapsule\*>(PyCapsule\_GetPointer(capsule, "arrow::Buffer"));

}

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

// pandas 0.x DataFrame conversion internals

using internal::arrow\_traits;

using internal::npy\_traits;

template <typename T>

struct WrapBytes {};

template <>

struct WrapBytes<StringType> {

static inline PyObject\* Wrap(const char\* data, int64\_t length) {

return PyUnicode\_FromStringAndSize(data, length);

}

};

template <>

struct WrapBytes<LargeStringType> {

static inline PyObject\* Wrap(const char\* data, int64\_t length) {

return PyUnicode\_FromStringAndSize(data, length);

}

};

template <>

struct WrapBytes<StringViewType> {

static inline PyObject\* Wrap(const char\* data, int64\_t length) {

return PyUnicode\_FromStringAndSize(data, length);

}

};

template <>

struct WrapBytes<BinaryType> {

static inline PyObject\* Wrap(const char\* data, int64\_t length) {

return PyBytes\_FromStringAndSize(data, length);

}

};

template <>

struct WrapBytes<LargeBinaryType> {

static inline PyObject\* Wrap(const char\* data, int64\_t length) {

return PyBytes\_FromStringAndSize(data, length);

}

};

template <>

struct WrapBytes<BinaryViewType> {

static inline PyObject\* Wrap(const char\* data, int64\_t length) {

return PyBytes\_FromStringAndSize(data, length);

}

};

template <>

struct WrapBytes<FixedSizeBinaryType> {

static inline PyObject\* Wrap(const char\* data, int64\_t length) {

return PyBytes\_FromStringAndSize(data, length);

}

};

static inline bool ListTypeSupported(const DataType& type) {

switch (type.id()) {

case Type::BOOL:

case Type::UINT8:

case Type::INT8:

case Type::UINT16:

case Type::INT16:

case Type::UINT32:

case Type::INT32:

case Type::INT64:

case Type::UINT64:

case Type::HALF\_FLOAT:

case Type::FLOAT:

case Type::DOUBLE:

case Type::DECIMAL128:

case Type::DECIMAL256:

case Type::BINARY:

case Type::LARGE\_BINARY:

case Type::STRING:

case Type::LARGE\_STRING:

case Type::DATE32:

case Type::DATE64:

case Type::STRUCT:

case Type::MAP:

case Type::TIME32:

case Type::TIME64:

case Type::TIMESTAMP:

case Type::DURATION:

case Type::DICTIONARY:

case Type::INTERVAL\_MONTH\_DAY\_NANO:

case Type::NA: // empty list

// The above types are all supported.

return true;

case Type::FIXED\_SIZE\_LIST:

case Type::LIST:

case Type::LARGE\_LIST:

case Type::LIST\_VIEW:

case Type::LARGE\_LIST\_VIEW: {

const auto& list\_type = checked\_cast<const BaseListType&>(type);

return ListTypeSupported(\*list\_type.value\_type());

}

case Type::EXTENSION: {

const auto& ext = checked\_cast<const ExtensionType&>(\*type.GetSharedPtr());

return ListTypeSupported(\*(ext.storage\_type()));

}

default:

break;

}

return false;

}

Status CapsulizeArray(const std::shared\_ptr<Array>& arr, PyObject\*\* out) {

auto capsule = new ArrayCapsule{{arr}};

\*out = PyCapsule\_New(reinterpret\_cast<void\*>(capsule), "arrow::Array",

&ArrayCapsule\_Destructor);

if (\*out == nullptr) {

delete capsule;

RETURN\_IF\_PYERROR();

}

return Status::OK();

}

Status CapsulizeBuffer(const std::shared\_ptr<Buffer>& buffer, PyObject\*\* out) {

auto capsule = new BufferCapsule{{buffer}};

\*out = PyCapsule\_New(reinterpret\_cast<void\*>(capsule), "arrow::Buffer",

&BufferCapsule\_Destructor);

if (\*out == nullptr) {

delete capsule;

RETURN\_IF\_PYERROR();

}

return Status::OK();

}

Status SetNdarrayBase(PyArrayObject\* arr, PyObject\* base) {

if (PyArray\_SetBaseObject(arr, base) == -1) {

// Error occurred, trust that SetBaseObject sets the error state

Py\_XDECREF(base);

RETURN\_IF\_PYERROR();

}

return Status::OK();

}

Status SetBufferBase(PyArrayObject\* arr, const std::shared\_ptr<Buffer>& buffer) {

PyObject\* base;

RETURN\_NOT\_OK(CapsulizeBuffer(buffer, &base));

return SetNdarrayBase(arr, base);

}

inline void set\_numpy\_metadata(int type, const DataType\* datatype, PyArray\_Descr\* out) {

auto metadata =

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

if (type == NPY\_DATETIME) {

if (datatype->id() == Type::TIMESTAMP) {

const auto& timestamp\_type = checked\_cast<const TimestampType&>(\*datatype);

metadata->meta.base = internal::NumPyFrequency(timestamp\_type.unit());

} else {

DCHECK(false) << "NPY\_DATETIME views only supported for Arrow TIMESTAMP types";

}

} else if (type == NPY\_TIMEDELTA) {

DCHECK\_EQ(datatype->id(), Type::DURATION);

const auto& duration\_type = checked\_cast<const DurationType&>(\*datatype);

metadata->meta.base = internal::NumPyFrequency(duration\_type.unit());

}

}

Status PyArray\_NewFromPool(int nd, npy\_intp\* dims, PyArray\_Descr\* descr, MemoryPool\* pool,

PyObject\*\* out) {

// ARROW-6570: Allocate memory from MemoryPool for a couple reasons

//

// \* Track allocations

// \* Get better performance through custom allocators

int64\_t total\_size = PyDataType\_ELSIZE(descr);

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

total\_size \*= dims[i];

}

ARROW\_ASSIGN\_OR\_RAISE(auto buffer, AllocateBuffer(total\_size, pool));

\*out = PyArray\_NewFromDescr(&PyArray\_Type, descr, nd, dims,

/\*strides=\*/nullptr,

/\*data=\*/buffer->mutable\_data(),

/\*flags=\*/NPY\_ARRAY\_CARRAY | NPY\_ARRAY\_WRITEABLE,

/\*obj=\*/nullptr);

if (\*out == nullptr) {

RETURN\_IF\_PYERROR();

// Trust that error set if NULL returned

}

return SetBufferBase(reinterpret\_cast<PyArrayObject\*>(\*out), std::move(buffer));

}

template <typename T = void>

inline const T\* GetPrimitiveValues(const Array& arr) {

if (arr.length() == 0) {

return nullptr;

}

const int elsize = arr.type()->byte\_width();

const auto& prim\_arr = checked\_cast<const PrimitiveArray&>(arr);

return reinterpret\_cast<const T\*>(prim\_arr.values()->data() + arr.offset() \* elsize);

}

Status MakeNumPyView(std::shared\_ptr<Array> arr, PyObject\* py\_ref, int npy\_type, int ndim,

npy\_intp\* dims, PyObject\*\* out) {

PyAcquireGIL lock;

PyArray\_Descr\* descr = internal::GetSafeNumPyDtype(npy\_type);

set\_numpy\_metadata(npy\_type, arr->type().get(), descr);

PyObject\* result = PyArray\_NewFromDescr(

&PyArray\_Type, descr, ndim, dims, /\*strides=\*/nullptr,

const\_cast<void\*>(GetPrimitiveValues(\*arr)), /\*flags=\*/0, nullptr);

PyArrayObject\* np\_arr = reinterpret\_cast<PyArrayObject\*>(result);

if (np\_arr == nullptr) {

// Error occurred, trust that error set

return Status::OK();

}

PyObject\* base;

if (py\_ref == nullptr) {

// Capsule will be owned by the ndarray, no incref necessary. See

// ARROW-1973

RETURN\_NOT\_OK(CapsulizeArray(arr, &base));

} else {

Py\_INCREF(py\_ref);

base = py\_ref;

}

RETURN\_NOT\_OK(SetNdarrayBase(np\_arr, base));

// Do not allow Arrow data to be mutated

PyArray\_CLEARFLAGS(np\_arr, NPY\_ARRAY\_WRITEABLE);

\*out = result;

return Status::OK();

}

class PandasWriter {

public:

enum type {

OBJECT,

UINT8,

INT8,

UINT16,

INT16,

UINT32,

INT32,

UINT64,

INT64,

HALF\_FLOAT,

FLOAT,

DOUBLE,

BOOL,

DATETIME\_DAY,

DATETIME\_SECOND,

DATETIME\_MILLI,

DATETIME\_MICRO,

DATETIME\_NANO,

DATETIME\_SECOND\_TZ,

DATETIME\_MILLI\_TZ,

DATETIME\_MICRO\_TZ,

DATETIME\_NANO\_TZ,

TIMEDELTA\_SECOND,

TIMEDELTA\_MILLI,

TIMEDELTA\_MICRO,

TIMEDELTA\_NANO,

CATEGORICAL,

EXTENSION

};

PandasWriter(const PandasOptions& options, int64\_t num\_rows, int num\_columns)

: options\_(options), num\_rows\_(num\_rows), num\_columns\_(num\_columns) {

PyAcquireGIL lock;

internal::InitPandasStaticData();

}

virtual ~PandasWriter() {}

void SetBlockData(PyObject\* arr) {

block\_arr\_.reset(arr);

block\_data\_ =

reinterpret\_cast<uint8\_t\*>(PyArray\_DATA(reinterpret\_cast<PyArrayObject\*>(arr)));

}

/// \brief Either copy or wrap single array to create pandas-compatible array

/// for Series or DataFrame. num\_columns\_ can only be 1. Will try to zero

/// copy if possible (or error if not possible and zero\_copy\_only=True)

virtual Status TransferSingle(std::shared\_ptr<ChunkedArray> data, PyObject\* py\_ref) = 0;

/// \brief Copy ChunkedArray into a multi-column block

virtual Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) = 0;

Status EnsurePlacementAllocated() {

std::lock\_guard<std::mutex> guard(allocation\_lock\_);

if (placement\_data\_ != nullptr) {

return Status::OK();

}

PyAcquireGIL lock;

npy\_intp placement\_dims[1] = {num\_columns\_};

PyObject\* placement\_arr = PyArray\_SimpleNew(1, placement\_dims, NPY\_INT64);

RETURN\_IF\_PYERROR();

placement\_arr\_.reset(placement\_arr);

placement\_data\_ = reinterpret\_cast<int64\_t\*>(

PyArray\_DATA(reinterpret\_cast<PyArrayObject\*>(placement\_arr)));

return Status::OK();

}

Status EnsureAllocated() {

std::lock\_guard<std::mutex> guard(allocation\_lock\_);

if (block\_data\_ != nullptr) {

return Status::OK();

}

RETURN\_NOT\_OK(Allocate());

return Status::OK();

}

virtual bool CanZeroCopy(const ChunkedArray& data) const { return false; }

virtual Status Write(std::shared\_ptr<ChunkedArray> data, int64\_t abs\_placement,

int64\_t rel\_placement) {

RETURN\_NOT\_OK(EnsurePlacementAllocated());

if (num\_columns\_ == 1 && options\_.allow\_zero\_copy\_blocks) {

RETURN\_NOT\_OK(TransferSingle(data, /\*py\_ref=\*/nullptr));

} else {

RETURN\_NOT\_OK(

CheckNoZeroCopy("Cannot do zero copy conversion into "

"multi-column DataFrame block"));

RETURN\_NOT\_OK(EnsureAllocated());

RETURN\_NOT\_OK(CopyInto(data, rel\_placement));

}

placement\_data\_[rel\_placement] = abs\_placement;

return Status::OK();

}

virtual Status GetDataFrameResult(PyObject\*\* out) {

PyObject\* result = PyDict\_New();

RETURN\_IF\_PYERROR();

PyObject\* block;

RETURN\_NOT\_OK(GetResultBlock(&block));

PyDict\_SetItemString(result, "block", block);

PyDict\_SetItemString(result, "placement", placement\_arr\_.obj());

RETURN\_NOT\_OK(AddResultMetadata(result));

\*out = result;

return Status::OK();

}

// Caller steals the reference to this object

virtual Status GetSeriesResult(PyObject\*\* out) {

RETURN\_NOT\_OK(MakeBlock1D());

// Caller owns the object now

\*out = block\_arr\_.detach();

return Status::OK();

}

protected:

virtual Status AddResultMetadata(PyObject\* result) { return Status::OK(); }

Status MakeBlock1D() {

// For Series or for certain DataFrame block types, we need to shape to a

// 1D array when there is only one column

PyAcquireGIL lock;

DCHECK\_EQ(1, num\_columns\_);

npy\_intp new\_dims[1] = {static\_cast<npy\_intp>(num\_rows\_)};

PyArray\_Dims dims;

dims.ptr = new\_dims;

dims.len = 1;

PyObject\* reshaped = PyArray\_Newshape(

reinterpret\_cast<PyArrayObject\*>(block\_arr\_.obj()), &dims, NPY\_ANYORDER);

RETURN\_IF\_PYERROR();

// ARROW-8801: Here a PyArrayObject is created that is not being managed by

// any OwnedRef object. This object is then put in the resulting object

// with PyDict\_SetItemString, which increments the reference count, so a

// memory leak ensues. There are several ways to fix the memory leak but a

// simple one is to put the reshaped 1D block array in this OwnedRefNoGIL

// so it will be correctly decref'd when this class is destructed.

block\_arr\_.reset(reshaped);

return Status::OK();

}

virtual Status GetResultBlock(PyObject\*\* out) {

\*out = block\_arr\_.obj();

return Status::OK();

}

Status CheckNoZeroCopy(const std::string& message) {

if (options\_.zero\_copy\_only) {

return Status::Invalid(message);

}

return Status::OK();

}

Status CheckNotZeroCopyOnly(const ChunkedArray& data) {

if (options\_.zero\_copy\_only) {

return Status::Invalid("Needed to copy ", data.num\_chunks(), " chunks with ",

data.null\_count(), " nulls, but zero\_copy\_only was True");

}

return Status::OK();

}

virtual Status Allocate() {

return Status::NotImplemented("Override Allocate in subclasses");

}

Status AllocateNDArray(int npy\_type, int ndim = 2) {

PyAcquireGIL lock;

PyObject\* block\_arr = nullptr;

npy\_intp block\_dims[2] = {0, 0};

if (ndim == 2) {

block\_dims[0] = num\_columns\_;

block\_dims[1] = num\_rows\_;

} else {

block\_dims[0] = num\_rows\_;

}

PyArray\_Descr\* descr = internal::GetSafeNumPyDtype(npy\_type);

if (PyDataType\_REFCHK(descr)) {

// ARROW-6876: if the array has refcounted items, let Numpy

// own the array memory so as to decref elements on array destruction

block\_arr = PyArray\_SimpleNewFromDescr(ndim, block\_dims, descr);

RETURN\_IF\_PYERROR();

} else {

RETURN\_NOT\_OK(

PyArray\_NewFromPool(ndim, block\_dims, descr, options\_.pool, &block\_arr));

}

SetBlockData(block\_arr);

return Status::OK();

}

void SetDatetimeUnit(NPY\_DATETIMEUNIT unit) {

PyAcquireGIL lock;

auto date\_dtype =

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

PyArray\_DESCR(reinterpret\_cast<PyArrayObject\*>(block\_arr\_.obj()))));

date\_dtype->meta.base = unit;

}

PandasOptions options\_;

std::mutex allocation\_lock\_;

int64\_t num\_rows\_;

int num\_columns\_;

OwnedRefNoGIL block\_arr\_;

uint8\_t\* block\_data\_ = nullptr;

// ndarray<int32>

OwnedRefNoGIL placement\_arr\_;

int64\_t\* placement\_data\_ = nullptr;

private:

ARROW\_DISALLOW\_COPY\_AND\_ASSIGN(PandasWriter);

};

template <typename InType, typename OutType>

inline void ConvertIntegerWithNulls(const PandasOptions& options,

const ChunkedArray& data, OutType\* out\_values) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = \*data.chunk(c);

const InType\* in\_values = GetPrimitiveValues<InType>(arr);

// Upcast to double, set NaN as appropriate

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

\*out\_values++ =

arr.IsNull(i) ? static\_cast<OutType>(NAN) : static\_cast<OutType>(in\_values[i]);

}

}

}

template <typename T>

inline void ConvertIntegerNoNullsSameType(const PandasOptions& options,

const ChunkedArray& data, T\* out\_values) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = \*data.chunk(c);

if (arr.length() > 0) {

const T\* in\_values = GetPrimitiveValues<T>(arr);

memcpy(out\_values, in\_values, sizeof(T) \* arr.length());

out\_values += arr.length();

}

}

}

template <typename InType, typename OutType>

inline void ConvertIntegerNoNullsCast(const PandasOptions& options,

const ChunkedArray& data, OutType\* out\_values) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = \*data.chunk(c);

const InType\* in\_values = GetPrimitiveValues<InType>(arr);

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

\*out\_values = in\_values[i];

}

}

}

template <typename T, typename Enable = void>

struct MemoizationTraits {

using Scalar = typename T::c\_type;

};

template <typename T>

struct MemoizationTraits<T, enable\_if\_has\_string\_view<T>> {

// For binary, we memoize string\_view as a scalar value to avoid having to

// unnecessarily copy the memory into the memo table data structure

using Scalar = std::string\_view;

};

// Generic Array -> PyObject\*\* converter that handles object deduplication, if

// requested

template <typename Type, typename WrapFunction>

inline Status ConvertAsPyObjects(const PandasOptions& options, const ChunkedArray& data,

WrapFunction&& wrap\_func, PyObject\*\* out\_values) {

using ArrayType = typename TypeTraits<Type>::ArrayType;

using Scalar = typename MemoizationTraits<Type>::Scalar;

auto convert\_chunks = [&](auto&& wrap\_func) -> Status {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = arrow::internal::checked\_cast<const ArrayType&>(\*data.chunk(c));

RETURN\_NOT\_OK(internal::WriteArrayObjects(arr, wrap\_func, out\_values));

out\_values += arr.length();

}

return Status::OK();

};

if (options.deduplicate\_objects) {

// GH-40316: only allocate a memo table if deduplication is enabled.

::arrow::internal::ScalarMemoTable<Scalar> memo\_table(options.pool);

std::vector<PyObject\*> unique\_values;

int32\_t memo\_size = 0;

auto WrapMemoized = [&](const Scalar& value, PyObject\*\* out\_values) {

int32\_t memo\_index;

RETURN\_NOT\_OK(memo\_table.GetOrInsert(value, &memo\_index));

if (memo\_index == memo\_size) {

// New entry

RETURN\_NOT\_OK(wrap\_func(value, out\_values));

unique\_values.push\_back(\*out\_values);

++memo\_size;

} else {

// Duplicate entry

Py\_INCREF(unique\_values[memo\_index]);

\*out\_values = unique\_values[memo\_index];

}

return Status::OK();

};

return convert\_chunks(std::move(WrapMemoized));

} else {

return convert\_chunks(std::forward<WrapFunction>(wrap\_func));

}

}

Status ConvertStruct(PandasOptions options, const ChunkedArray& data,

PyObject\*\* out\_values) {

if (data.num\_chunks() == 0) {

return Status::OK();

}

// ChunkedArray has at least one chunk

auto arr = checked\_cast<const StructArray\*>(data.chunk(0).get());

// Use it to cache the struct type and number of fields for all chunks

int32\_t num\_fields = arr->num\_fields();

auto array\_type = arr->type();

std::vector<OwnedRef> fields\_data(num\_fields \* data.num\_chunks());

OwnedRef dict\_item;

// See notes in MakeInnerOptions.

options = MakeInnerOptions(std::move(options));

// Don't blindly convert because timestamps in lists are handled differently.

options.timestamp\_as\_object = true;

for (int c = 0; c < data.num\_chunks(); c++) {

auto fields\_data\_offset = c \* num\_fields;

auto arr = checked\_cast<const StructArray\*>(data.chunk(c).get());

// Convert the struct arrays first

for (int32\_t i = 0; i < num\_fields; i++) {

auto field = arr->field(static\_cast<int>(i));

// In case the field is an extension array, use .storage() to convert to Pandas

if (field->type()->id() == Type::EXTENSION) {

const ExtensionArray& arr\_ext = checked\_cast<const ExtensionArray&>(\*field);

field = arr\_ext.storage();

}

RETURN\_NOT\_OK(ConvertArrayToPandas(options, field, nullptr,

fields\_data[i + fields\_data\_offset].ref()));

DCHECK(PyArray\_Check(fields\_data[i + fields\_data\_offset].obj()));

}

// Construct a dictionary for each row

const bool has\_nulls = data.null\_count() > 0;

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

if (has\_nulls && arr->IsNull(i)) {

Py\_INCREF(Py\_None);

\*out\_values = Py\_None;

} else {

// Build the new dict object for the row

dict\_item.reset(PyDict\_New());

RETURN\_IF\_PYERROR();

for (int32\_t field\_idx = 0; field\_idx < num\_fields; ++field\_idx) {

OwnedRef field\_value;

auto name = array\_type->field(static\_cast<int>(field\_idx))->name();

if (!arr->field(static\_cast<int>(field\_idx))->IsNull(i)) {

// Value exists in child array, obtain it

auto array = reinterpret\_cast<PyArrayObject\*>(

fields\_data[field\_idx + fields\_data\_offset].obj());

auto ptr = reinterpret\_cast<const char\*>(PyArray\_GETPTR1(array, i));

field\_value.reset(PyArray\_GETITEM(array, ptr));

RETURN\_IF\_PYERROR();

} else {

// Translate the Null to a None

Py\_INCREF(Py\_None);

field\_value.reset(Py\_None);

}

// PyDict\_SetItemString increments reference count

auto setitem\_result =

PyDict\_SetItemString(dict\_item.obj(), name.c\_str(), field\_value.obj());

RETURN\_IF\_PYERROR();

DCHECK\_EQ(setitem\_result, 0);

}

\*out\_values = dict\_item.obj();

// Grant ownership to the resulting array

Py\_INCREF(\*out\_values);

}

++out\_values;

}

}

return Status::OK();

}

Status DecodeDictionaries(MemoryPool\* pool, const std::shared\_ptr<DataType>& dense\_type,

ArrayVector\* arrays) {

compute::ExecContext ctx(pool);

compute::CastOptions options;

for (size\_t i = 0; i < arrays->size(); ++i) {

ARROW\_ASSIGN\_OR\_RAISE((\*arrays)[i],

compute::Cast(\*(\*arrays)[i], dense\_type, options, &ctx));

}

return Status::OK();

}

Status DecodeDictionaries(MemoryPool\* pool, const std::shared\_ptr<DataType>& dense\_type,

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

auto chunks = (\*array)->chunks();

RETURN\_NOT\_OK(DecodeDictionaries(pool, dense\_type, &chunks));

\*array = std::make\_shared<ChunkedArray>(std::move(chunks), dense\_type);

return Status::OK();

}

template <typename T>

enable\_if\_list\_like<T, Status> ConvertListsLike(PandasOptions options,

const ChunkedArray& data,

PyObject\*\* out\_values) {

using ListArrayT = typename TypeTraits<T>::ArrayType;

// Get column of underlying value arrays

ArrayVector value\_arrays;

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const ListArrayT&>(\*data.chunk(c));

// values() does not account for offsets, so we need to slice into it.

// We can't use Flatten(), because it removes the values behind a null list

// value, and that makes the offsets into original list values and our

// flattened\_values array different.

std::shared\_ptr<Array> flattened\_values = arr.values()->Slice(

arr.value\_offset(0), arr.value\_offset(arr.length()) - arr.value\_offset(0));

if (arr.value\_type()->id() == Type::EXTENSION) {

const auto& arr\_ext = checked\_cast<const ExtensionArray&>(\*flattened\_values);

value\_arrays.emplace\_back(arr\_ext.storage());

} else {

value\_arrays.emplace\_back(flattened\_values);

}

}

using ListArrayType = typename ListArrayT::TypeClass;

const auto& list\_type = checked\_cast<const ListArrayType&>(\*data.type());

auto value\_type = list\_type.value\_type();

if (value\_type->id() == Type::EXTENSION) {

value\_type = checked\_cast<const ExtensionType&>(\*value\_type).storage\_type();

}

auto flat\_column = std::make\_shared<ChunkedArray>(value\_arrays, value\_type);

options = MakeInnerOptions(std::move(options));

OwnedRefNoGIL owned\_numpy\_array;

RETURN\_NOT\_OK(ConvertChunkedArrayToPandas(options, flat\_column, nullptr,

owned\_numpy\_array.ref()));

PyObject\* numpy\_array = owned\_numpy\_array.obj();

DCHECK(PyArray\_Check(numpy\_array));

int64\_t chunk\_offset = 0;

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const ListArrayT&>(\*data.chunk(c));

const bool has\_nulls = data.null\_count() > 0;

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

if (has\_nulls && arr.IsNull(i)) {

Py\_INCREF(Py\_None);

\*out\_values = Py\_None;

} else {

// Need to subtract value\_offset(0) since the original chunk might be a slice

// into another array.

OwnedRef start(PyLong\_FromLongLong(arr.value\_offset(i) + chunk\_offset -

arr.value\_offset(0)));

OwnedRef end(PyLong\_FromLongLong(arr.value\_offset(i + 1) + chunk\_offset -

arr.value\_offset(0)));

OwnedRef slice(PySlice\_New(start.obj(), end.obj(), nullptr));

if (ARROW\_PREDICT\_FALSE(slice.obj() == nullptr)) {

// Fall out of loop, will return from RETURN\_IF\_PYERROR

break;

}

\*out\_values = PyObject\_GetItem(numpy\_array, slice.obj());

if (\*out\_values == nullptr) {

// Fall out of loop, will return from RETURN\_IF\_PYERROR

break;

}

}

++out\_values;

}

RETURN\_IF\_PYERROR();

chunk\_offset += arr.value\_offset(arr.length()) - arr.value\_offset(0);

}

return Status::OK();

}

// TODO GH-40579: optimize ListView conversion to avoid unnecessary copies

template <typename T>

enable\_if\_list\_view<T, Status> ConvertListsLike(PandasOptions options,

const ChunkedArray& data,

PyObject\*\* out\_values) {

using ListViewArrayType = typename TypeTraits<T>::ArrayType;

using NonViewType =

std::conditional\_t<T::type\_id == Type::LIST\_VIEW, ListType, LargeListType>;

using NonViewClass = typename TypeTraits<NonViewType>::ArrayType;

ArrayVector list\_arrays;

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const ListViewArrayType&>(\*data.chunk(c));

ARROW\_ASSIGN\_OR\_RAISE(auto non\_view\_array,

NonViewClass::FromListView(arr, options.pool));

list\_arrays.emplace\_back(non\_view\_array);

}

auto chunked\_array = std::make\_shared<ChunkedArray>(list\_arrays);

return ConvertListsLike<NonViewType>(options, \*chunked\_array, out\_values);

}

template <typename F1, typename F2, typename F3>

Status ConvertMapHelper(F1 resetRow, F2 addPairToRow, F3 stealRow,

const ChunkedArray& data, PyArrayObject\* py\_keys,

PyArrayObject\* py\_items,

// needed for null checks in items

const std::vector<std::shared\_ptr<Array>> item\_arrays,

PyObject\*\* out\_values) {

OwnedRef key\_value;

OwnedRef item\_value;

int64\_t chunk\_offset = 0;

for (int c = 0; c < data.num\_chunks(); ++c) {

const auto& arr = checked\_cast<const MapArray&>(\*data.chunk(c));

const bool has\_nulls = data.null\_count() > 0;

// Make a list of key/item pairs for each row in array

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

if (has\_nulls && arr.IsNull(i)) {

Py\_INCREF(Py\_None);

\*out\_values = Py\_None;

} else {

int64\_t entry\_offset = arr.value\_offset(i);

int64\_t num\_pairs = arr.value\_offset(i + 1) - entry\_offset;

// Build the new list object for the row of Python pairs

RETURN\_NOT\_OK(resetRow(num\_pairs));

// Add each key/item pair in the row

for (int64\_t j = 0; j < num\_pairs; ++j) {

// Get key value, key is non-nullable for a valid row

auto ptr\_key = reinterpret\_cast<const char\*>(

PyArray\_GETPTR1(py\_keys, chunk\_offset + entry\_offset + j));

key\_value.reset(PyArray\_GETITEM(py\_keys, ptr\_key));

RETURN\_IF\_PYERROR();

if (item\_arrays[c]->IsNull(entry\_offset + j)) {

// Translate the Null to a None

Py\_INCREF(Py\_None);

item\_value.reset(Py\_None);

} else {

// Get valid value from item array

auto ptr\_item = reinterpret\_cast<const char\*>(

PyArray\_GETPTR1(py\_items, chunk\_offset + entry\_offset + j));

item\_value.reset(PyArray\_GETITEM(py\_items, ptr\_item));

RETURN\_IF\_PYERROR();

}

// Add the key/item pair to the row

RETURN\_NOT\_OK(addPairToRow(j, key\_value, item\_value));

}

// Pass ownership to the resulting array

\*out\_values = stealRow();

}

++out\_values;

}

RETURN\_IF\_PYERROR();

chunk\_offset += arr.values()->length();

}

return Status::OK();

}

// A more helpful error message around TypeErrors that may stem from unhashable keys

Status CheckMapAsPydictsTypeError() {

if (ARROW\_PREDICT\_TRUE(!PyErr\_Occurred())) {

return Status::OK();

}

if (PyErr\_ExceptionMatches(PyExc\_TypeError)) {

// Modify the error string directly, so it is re-raised

// with our additional info.

//

// There are not many interesting things happening when this

// is hit. This is intended to only be called directly after

// PyDict\_SetItem, where a finite set of errors could occur.

PyObject \*type, \*value, \*traceback;

PyErr\_Fetch(&type, &value, &traceback);

std::string message;

RETURN\_NOT\_OK(internal::PyObject\_StdStringStr(value, &message));

message +=

". If keys are not hashable, then you must use the option "

"[maps\_as\_pydicts=None (default)]";

// resets the error

PyErr\_SetString(PyExc\_TypeError, message.c\_str());

}

return ConvertPyError();

}

Status CheckForDuplicateKeys(bool error\_on\_duplicate\_keys, Py\_ssize\_t total\_dict\_len,

Py\_ssize\_t total\_raw\_len) {

if (total\_dict\_len < total\_raw\_len) {

const char\* message =

"[maps\_as\_pydicts] "

"After conversion of Arrow maps to pydicts, "

"detected data loss due to duplicate keys. "

"Original input length is [%lld], total converted pydict length is [%lld].";

std::array<char, 256> buf;

std::snprintf(buf.data(), buf.size(), message, total\_raw\_len, total\_dict\_len);

if (error\_on\_duplicate\_keys) {

return Status::UnknownError(buf.data());

} else {

ARROW\_LOG(WARNING) << buf.data();

}

}

return Status::OK();

}

Status ConvertMap(PandasOptions options, const ChunkedArray& data,

PyObject\*\* out\_values) {

// Get columns of underlying key/item arrays

std::vector<std::shared\_ptr<Array>> key\_arrays;

std::vector<std::shared\_ptr<Array>> item\_arrays;

for (int c = 0; c < data.num\_chunks(); ++c) {

const auto& map\_arr = checked\_cast<const MapArray&>(\*data.chunk(c));

key\_arrays.emplace\_back(map\_arr.keys());

item\_arrays.emplace\_back(map\_arr.items());

}

const auto& map\_type = checked\_cast<const MapType&>(\*data.type());

auto key\_type = map\_type.key\_type();

auto item\_type = map\_type.item\_type();

// ARROW-6899: Convert dictionary-encoded children to dense instead of

// failing below. A more efficient conversion than this could be done later

if (key\_type->id() == Type::DICTIONARY) {

auto dense\_type = checked\_cast<const DictionaryType&>(\*key\_type).value\_type();

RETURN\_NOT\_OK(DecodeDictionaries(options.pool, dense\_type, &key\_arrays));

key\_type = dense\_type;

}

if (item\_type->id() == Type::DICTIONARY) {

auto dense\_type = checked\_cast<const DictionaryType&>(\*item\_type).value\_type();

RETURN\_NOT\_OK(DecodeDictionaries(options.pool, dense\_type, &item\_arrays));

item\_type = dense\_type;

}

// See notes in MakeInnerOptions.

options = MakeInnerOptions(std::move(options));

// Don't blindly convert because timestamps in lists are handled differently.

options.timestamp\_as\_object = true;

auto flat\_keys = std::make\_shared<ChunkedArray>(key\_arrays, key\_type);

auto flat\_items = std::make\_shared<ChunkedArray>(item\_arrays, item\_type);

OwnedRefNoGIL owned\_numpy\_keys;

RETURN\_NOT\_OK(

ConvertChunkedArrayToPandas(options, flat\_keys, nullptr, owned\_numpy\_keys.ref()));

OwnedRefNoGIL owned\_numpy\_items;

RETURN\_NOT\_OK(

ConvertChunkedArrayToPandas(options, flat\_items, nullptr, owned\_numpy\_items.ref()));

PyArrayObject\* py\_keys = reinterpret\_cast<PyArrayObject\*>(owned\_numpy\_keys.obj());

PyArrayObject\* py\_items = reinterpret\_cast<PyArrayObject\*>(owned\_numpy\_items.obj());

if (options.maps\_as\_pydicts == MapConversionType::DEFAULT) {

// The default behavior to express an Arrow MAP as a list of [(key, value), ...] pairs

OwnedRef list\_item;

return ConvertMapHelper(

[&list\_item](int64\_t num\_pairs) {

list\_item.reset(PyList\_New(num\_pairs));

return CheckPyError();

},

[&list\_item](int64\_t idx, OwnedRef& key\_value, OwnedRef& item\_value) {

PyList\_SET\_ITEM(list\_item.obj(), idx,

PyTuple\_Pack(2, key\_value.obj(), item\_value.obj()));

return CheckPyError();

},

[&list\_item] { return list\_item.detach(); }, data, py\_keys, py\_items, item\_arrays,

out\_values);

} else {

// Use a native pydict

OwnedRef dict\_item;

Py\_ssize\_t total\_dict\_len{0};

Py\_ssize\_t total\_raw\_len{0};

bool error\_on\_duplicate\_keys;

if (options.maps\_as\_pydicts == MapConversionType::LOSSY) {

error\_on\_duplicate\_keys = false;

} else if (options.maps\_as\_pydicts == MapConversionType::STRICT\_) {

error\_on\_duplicate\_keys = true;

} else {

auto val = std::underlying\_type\_t<MapConversionType>(options.maps\_as\_pydicts);

return Status::UnknownError("Received unknown option for maps\_as\_pydicts: " +

std::to\_string(val));

}

auto status = ConvertMapHelper(

[&dict\_item, &total\_raw\_len](int64\_t num\_pairs) {

total\_raw\_len += num\_pairs;

dict\_item.reset(PyDict\_New());

return CheckPyError();

},

[&dict\_item]([[maybe\_unused]] int64\_t idx, OwnedRef& key\_value,

OwnedRef& item\_value) {

auto setitem\_result =

PyDict\_SetItem(dict\_item.obj(), key\_value.obj(), item\_value.obj());

ARROW\_RETURN\_NOT\_OK(CheckMapAsPydictsTypeError());

// returns -1 if there are internal errors around hashing/resizing

return setitem\_result == 0 ? Status::OK()

: Status::UnknownError(

"[maps\_as\_pydicts] "

"Unexpected failure inserting Arrow (key, "

"value) pair into Python dict");

},

[&dict\_item, &total\_dict\_len] {

total\_dict\_len += PyDict\_Size(dict\_item.obj());

return dict\_item.detach();

},

data, py\_keys, py\_items, item\_arrays, out\_values);

ARROW\_RETURN\_NOT\_OK(status);

// If there were no errors generating the pydicts,

// then check if we detected any data loss from duplicate keys.

return CheckForDuplicateKeys(error\_on\_duplicate\_keys, total\_dict\_len, total\_raw\_len);

}

}

template <typename InType, typename OutType>

inline void ConvertNumericNullable(const ChunkedArray& data, InType na\_value,

OutType\* out\_values) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = \*data.chunk(c);

const InType\* in\_values = GetPrimitiveValues<InType>(arr);

if (arr.null\_count() > 0) {

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

\*out\_values++ = arr.IsNull(i) ? na\_value : in\_values[i];

}

} else {

memcpy(out\_values, in\_values, sizeof(InType) \* arr.length());

out\_values += arr.length();

}

}

}

template <typename InType, typename OutType>

inline void ConvertNumericNullableCast(const ChunkedArray& data, InType na\_value,

OutType\* out\_values) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = \*data.chunk(c);

const InType\* in\_values = GetPrimitiveValues<InType>(arr);

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

\*out\_values++ = arr.IsNull(i) ? static\_cast<OutType>(na\_value)

: static\_cast<OutType>(in\_values[i]);

}

}

}

template <int NPY\_TYPE>

class TypedPandasWriter : public PandasWriter {

public:

using T = typename npy\_traits<NPY\_TYPE>::value\_type;

using PandasWriter::PandasWriter;

Status TransferSingle(std::shared\_ptr<ChunkedArray> data, PyObject\* py\_ref) override {

if (CanZeroCopy(\*data)) {

PyObject\* wrapped;

npy\_intp dims[2] = {static\_cast<npy\_intp>(num\_columns\_),

static\_cast<npy\_intp>(num\_rows\_)};

RETURN\_NOT\_OK(

MakeNumPyView(data->chunk(0), py\_ref, NPY\_TYPE, /\*ndim=\*/2, dims, &wrapped));

SetBlockData(wrapped);

return Status::OK();

} else {

RETURN\_NOT\_OK(CheckNotZeroCopyOnly(\*data));

RETURN\_NOT\_OK(EnsureAllocated());

return CopyInto(data, /\*rel\_placement=\*/0);

}

}

Status CheckTypeExact(const DataType& type, Type::type expected) {

if (type.id() != expected) {

// TODO(wesm): stringify NumPy / pandas type

return Status::NotImplemented("Cannot write Arrow data of type ", type.ToString());

}

return Status::OK();

}

T\* GetBlockColumnStart(int64\_t rel\_placement) {

return reinterpret\_cast<T\*>(block\_data\_) + rel\_placement \* num\_rows\_;

}

protected:

Status Allocate() override { return AllocateNDArray(NPY\_TYPE); }

};

struct ObjectWriterVisitor {

const PandasOptions& options;

const ChunkedArray& data;

PyObject\*\* out\_values;

Status Visit(const NullType& type) {

for (int c = 0; c < data.num\_chunks(); c++) {

std::shared\_ptr<Array> arr = data.chunk(c);

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

// All values are null

Py\_INCREF(Py\_None);

\*out\_values = Py\_None;

++out\_values;

}

}

return Status::OK();

}

Status Visit(const BooleanType& type) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const BooleanArray&>(\*data.chunk(c));

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

if (arr.IsNull(i)) {

Py\_INCREF(Py\_None);

\*out\_values++ = Py\_None;

} else if (arr.Value(i)) {

// True

Py\_INCREF(Py\_True);

\*out\_values++ = Py\_True;

} else {

// False

Py\_INCREF(Py\_False);

\*out\_values++ = Py\_False;

}

}

}

return Status::OK();

}

template <typename Type>

enable\_if\_integer<Type, Status> Visit(const Type& type) {

using T = typename Type::c\_type;

auto WrapValue = [](T value, PyObject\*\* out) {

\*out = std::is\_signed<T>::value ? PyLong\_FromLongLong(value)

: PyLong\_FromUnsignedLongLong(value);

RETURN\_IF\_PYERROR();

return Status::OK();

};

return ConvertAsPyObjects<Type>(options, data, WrapValue, out\_values);

}

template <typename Type>

enable\_if\_t<is\_base\_binary\_type<Type>::value || is\_binary\_view\_like\_type<Type>::value ||

is\_fixed\_size\_binary\_type<Type>::value,

Status>

Visit(const Type& type) {

auto WrapValue = [](const std::string\_view& view, PyObject\*\* out) {

\*out = WrapBytes<Type>::Wrap(view.data(), view.length());

if (\*out == nullptr) {

PyErr\_Clear();

return Status::UnknownError("Wrapping ", view, " failed");

}

return Status::OK();

};

return ConvertAsPyObjects<Type>(options, data, WrapValue, out\_values);

}

template <typename Type>

enable\_if\_date<Type, Status> Visit(const Type& type) {

auto WrapValue = [](typename Type::c\_type value, PyObject\*\* out) {

RETURN\_NOT\_OK(internal::PyDate\_from\_int(value, Type::UNIT, out));

RETURN\_IF\_PYERROR();

return Status::OK();

};

return ConvertAsPyObjects<Type>(options, data, WrapValue, out\_values);

}

template <typename Type>

enable\_if\_time<Type, Status> Visit(const Type& type) {

const TimeUnit::type unit = type.unit();

auto WrapValue = [unit](typename Type::c\_type value, PyObject\*\* out) {

RETURN\_NOT\_OK(internal::PyTime\_from\_int(value, unit, out));

RETURN\_IF\_PYERROR();

return Status::OK();

};

return ConvertAsPyObjects<Type>(options, data, WrapValue, out\_values);

}

template <typename Type>

enable\_if\_timestamp<Type, Status> Visit(const Type& type) {

const TimeUnit::type unit = type.unit();

OwnedRef tzinfo;

auto ConvertTimezoneNaive = [&](typename Type::c\_type value, PyObject\*\* out) {

RETURN\_NOT\_OK(internal::PyDateTime\_from\_int(value, unit, out));

RETURN\_IF\_PYERROR();

return Status::OK();

};

auto ConvertTimezoneAware = [&](typename Type::c\_type value, PyObject\*\* out) {

PyObject\* naive\_datetime;

RETURN\_NOT\_OK(ConvertTimezoneNaive(value, &naive\_datetime));

// convert the timezone naive datetime object to timezone aware

// two step conversion of the datetime mimics Python's code:

// dt.replace(tzinfo=datetime.timezone.utc).astimezone(tzinfo)

// first step: replacing timezone with timezone.utc (replace method)

OwnedRef args(PyTuple\_New(0));

OwnedRef keywords(PyDict\_New());

PyDict\_SetItemString(keywords.obj(), "tzinfo", PyDateTime\_TimeZone\_UTC);

OwnedRef naive\_datetime\_replace(PyObject\_GetAttrString(naive\_datetime, "replace"));

OwnedRef datetime\_utc(

PyObject\_Call(naive\_datetime\_replace.obj(), args.obj(), keywords.obj()));

// second step: adjust the datetime to tzinfo timezone (astimezone method)

\*out = PyObject\_CallMethod(datetime\_utc.obj(), "astimezone", "O", tzinfo.obj());

// the timezone naive object is no longer required

Py\_DECREF(naive\_datetime);

RETURN\_IF\_PYERROR();

return Status::OK();

};

if (!type.timezone().empty() && !options.ignore\_timezone) {

// convert timezone aware

PyObject\* tzobj;

ARROW\_ASSIGN\_OR\_RAISE(tzobj, internal::StringToTzinfo(type.timezone()));

tzinfo.reset(tzobj);

RETURN\_IF\_PYERROR();

RETURN\_NOT\_OK(

ConvertAsPyObjects<Type>(options, data, ConvertTimezoneAware, out\_values));

} else {

// convert timezone naive

RETURN\_NOT\_OK(

ConvertAsPyObjects<Type>(options, data, ConvertTimezoneNaive, out\_values));

}

return Status::OK();

}

template <typename Type>

enable\_if\_t<std::is\_same<Type, MonthDayNanoIntervalType>::value, Status> Visit(

const Type& type) {

OwnedRef args(PyTuple\_New(0));

OwnedRef kwargs(PyDict\_New());

RETURN\_IF\_PYERROR();

auto to\_date\_offset = [&](const MonthDayNanoIntervalType::MonthDayNanos& interval,

PyObject\*\* out) {

DCHECK(internal::BorrowPandasDataOffsetType() != nullptr);

// DateOffset objects do not add nanoseconds component to pd.Timestamp.

// as of Pandas 1.3.3

// (https://github.com/pandas-dev/pandas/issues/43892).

// So convert microseconds and remainder to preserve data

// but give users more expected results.

int64\_t microseconds = interval.nanoseconds / 1000;

int64\_t nanoseconds;

if (interval.nanoseconds >= 0) {

nanoseconds = interval.nanoseconds % 1000;

} else {

nanoseconds = -((-interval.nanoseconds) % 1000);

}

PyDict\_SetItemString(kwargs.obj(), "months", PyLong\_FromLong(interval.months));

PyDict\_SetItemString(kwargs.obj(), "days", PyLong\_FromLong(interval.days));

PyDict\_SetItemString(kwargs.obj(), "microseconds",

PyLong\_FromLongLong(microseconds));

PyDict\_SetItemString(kwargs.obj(), "nanoseconds", PyLong\_FromLongLong(nanoseconds));

\*out =

PyObject\_Call(internal::BorrowPandasDataOffsetType(), args.obj(), kwargs.obj());

RETURN\_IF\_PYERROR();

return Status::OK();

};

return ConvertAsPyObjects<MonthDayNanoIntervalType>(options, data, to\_date\_offset,

out\_values);

}

Status Visit(const Decimal32Type& type) {

return Status::NotImplemented("Decimal32 type not yet implemented");

}

Status Visit(const Decimal64Type& type) {

return Status::NotImplemented("Decimal64 type not yet implemented");

}

Status Visit(const Decimal128Type& type) {

OwnedRef decimal;

OwnedRef Decimal;

RETURN\_NOT\_OK(internal::ImportModule("decimal", &decimal));

RETURN\_NOT\_OK(internal::ImportFromModule(decimal.obj(), "Decimal", &Decimal));

PyObject\* decimal\_constructor = Decimal.obj();

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const arrow::Decimal128Array&>(\*data.chunk(c));

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

if (arr.IsNull(i)) {

Py\_INCREF(Py\_None);

\*out\_values++ = Py\_None;

} else {

\*out\_values++ =

internal::DecimalFromString(decimal\_constructor, arr.FormatValue(i));

RETURN\_IF\_PYERROR();

}

}

}

return Status::OK();

}

Status Visit(const Decimal256Type& type) {

OwnedRef decimal;

OwnedRef Decimal;

RETURN\_NOT\_OK(internal::ImportModule("decimal", &decimal));

RETURN\_NOT\_OK(internal::ImportFromModule(decimal.obj(), "Decimal", &Decimal));

PyObject\* decimal\_constructor = Decimal.obj();

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const arrow::Decimal256Array&>(\*data.chunk(c));

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

if (arr.IsNull(i)) {

Py\_INCREF(Py\_None);

\*out\_values++ = Py\_None;

} else {

\*out\_values++ =

internal::DecimalFromString(decimal\_constructor, arr.FormatValue(i));

RETURN\_IF\_PYERROR();

}

}

}

return Status::OK();

}

template <typename T>

enable\_if\_t<is\_list\_like\_type<T>::value || is\_list\_view\_type<T>::value, Status> Visit(

const T& type) {

if (!ListTypeSupported(\*type.value\_type())) {

return Status::NotImplemented(

"Not implemented type for conversion from List to Pandas: ",

type.value\_type()->ToString());

}

return ConvertListsLike<T>(options, data, out\_values);

}

Status Visit(const MapType& type) { return ConvertMap(options, data, out\_values); }

Status Visit(const StructType& type) {

return ConvertStruct(options, data, out\_values);

}

template <typename Type>

enable\_if\_t<is\_floating\_type<Type>::value ||

std::is\_same<DictionaryType, Type>::value ||

std::is\_same<DurationType, Type>::value ||

std::is\_same<RunEndEncodedType, Type>::value ||

std::is\_same<ExtensionType, Type>::value ||

(std::is\_base\_of<IntervalType, Type>::value &&

!std::is\_same<MonthDayNanoIntervalType, Type>::value) ||

std::is\_base\_of<UnionType, Type>::value,

Status>

Visit(const Type& type) {

return Status::NotImplemented("No implemented conversion to object dtype: ",

type.ToString());

}

};

class ObjectWriter : public TypedPandasWriter<NPY\_OBJECT> {

public:

using TypedPandasWriter<NPY\_OBJECT>::TypedPandasWriter;

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

PyAcquireGIL lock;

ObjectWriterVisitor visitor{this->options\_, \*data,

this->GetBlockColumnStart(rel\_placement)};

return VisitTypeInline(\*data->type(), &visitor);

}

};

static inline bool IsNonNullContiguous(const ChunkedArray& data) {

return data.num\_chunks() == 1 && data.null\_count() == 0;

}

template <int NPY\_TYPE>

class IntWriter : public TypedPandasWriter<NPY\_TYPE> {

public:

using ArrowType = typename npy\_traits<NPY\_TYPE>::TypeClass;

using TypedPandasWriter<NPY\_TYPE>::TypedPandasWriter;

bool CanZeroCopy(const ChunkedArray& data) const override {

return IsNonNullContiguous(data);

}

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

RETURN\_NOT\_OK(this->CheckTypeExact(\*data->type(), ArrowType::type\_id));

ConvertIntegerNoNullsSameType<typename ArrowType::c\_type>(

this->options\_, \*data, this->GetBlockColumnStart(rel\_placement));

return Status::OK();

}

};

template <int NPY\_TYPE>

class FloatWriter : public TypedPandasWriter<NPY\_TYPE> {

public:

using ArrowType = typename npy\_traits<NPY\_TYPE>::TypeClass;

using TypedPandasWriter<NPY\_TYPE>::TypedPandasWriter;

using T = typename ArrowType::c\_type;

bool CanZeroCopy(const ChunkedArray& data) const override {

return IsNonNullContiguous(data) && data.type()->id() == ArrowType::type\_id;

}

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

Type::type in\_type = data->type()->id();

auto out\_values = this->GetBlockColumnStart(rel\_placement);

#define INTEGER\_CASE(IN\_TYPE) \

ConvertIntegerWithNulls<IN\_TYPE, T>(this->options\_, \*data, out\_values); \

break;

switch (in\_type) {

case Type::UINT8:

INTEGER\_CASE(uint8\_t);

case Type::INT8:

INTEGER\_CASE(int8\_t);

case Type::UINT16:

INTEGER\_CASE(uint16\_t);

case Type::INT16:

INTEGER\_CASE(int16\_t);

case Type::UINT32:

INTEGER\_CASE(uint32\_t);

case Type::INT32:

INTEGER\_CASE(int32\_t);

case Type::UINT64:

INTEGER\_CASE(uint64\_t);

case Type::INT64:

INTEGER\_CASE(int64\_t);

case Type::HALF\_FLOAT:

ConvertNumericNullableCast(\*data, npy\_traits<NPY\_TYPE>::na\_sentinel, out\_values);

case Type::FLOAT:

ConvertNumericNullableCast(\*data, npy\_traits<NPY\_TYPE>::na\_sentinel, out\_values);

break;

case Type::DOUBLE:

ConvertNumericNullableCast(\*data, npy\_traits<NPY\_TYPE>::na\_sentinel, out\_values);

break;

default:

return Status::NotImplemented("Cannot write Arrow data of type ",

data->type()->ToString(),

" to a Pandas floating point block");

}

#undef INTEGER\_CASE

return Status::OK();

}

};

using UInt8Writer = IntWriter<NPY\_UINT8>;

using Int8Writer = IntWriter<NPY\_INT8>;

using UInt16Writer = IntWriter<NPY\_UINT16>;

using Int16Writer = IntWriter<NPY\_INT16>;

using UInt32Writer = IntWriter<NPY\_UINT32>;

using Int32Writer = IntWriter<NPY\_INT32>;

using UInt64Writer = IntWriter<NPY\_UINT64>;

using Int64Writer = IntWriter<NPY\_INT64>;

using Float16Writer = FloatWriter<NPY\_FLOAT16>;

using Float32Writer = FloatWriter<NPY\_FLOAT32>;

using Float64Writer = FloatWriter<NPY\_FLOAT64>;

class BoolWriter : public TypedPandasWriter<NPY\_BOOL> {

public:

using TypedPandasWriter<NPY\_BOOL>::TypedPandasWriter;

Status TransferSingle(std::shared\_ptr<ChunkedArray> data, PyObject\* py\_ref) override {

RETURN\_NOT\_OK(

CheckNoZeroCopy("Zero copy conversions not possible with "

"boolean types"));

RETURN\_NOT\_OK(EnsureAllocated());

return CopyInto(data, /\*rel\_placement=\*/0);

}

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

RETURN\_NOT\_OK(this->CheckTypeExact(\*data->type(), Type::BOOL));

auto out\_values = this->GetBlockColumnStart(rel\_placement);

for (int c = 0; c < data->num\_chunks(); c++) {

const auto& arr = checked\_cast<const BooleanArray&>(\*data->chunk(c));

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

\*out\_values++ = static\_cast<uint8\_t>(arr.Value(i));

}

}

return Status::OK();

}

};

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

// Date / timestamp types

template <typename T, int64\_t SHIFT>

inline void ConvertDatetime(const ChunkedArray& data, int64\_t\* out\_values) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = \*data.chunk(c);

const T\* in\_values = GetPrimitiveValues<T>(arr);

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

\*out\_values++ = arr.IsNull(i) ? kPandasTimestampNull

: (static\_cast<int64\_t>(in\_values[i]) \* SHIFT);

}

}

}

template <typename T, int SHIFT>

void ConvertDatesShift(const ChunkedArray& data, int64\_t\* out\_values) {

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = \*data.chunk(c);

const T\* in\_values = GetPrimitiveValues<T>(arr);

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

\*out\_values++ = arr.IsNull(i) ? kPandasTimestampNull

: static\_cast<int64\_t>(in\_values[i]) / SHIFT;

}

}

}

class DatetimeDayWriter : public TypedPandasWriter<NPY\_DATETIME> {

public:

using TypedPandasWriter<NPY\_DATETIME>::TypedPandasWriter;

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

int64\_t\* out\_values = this->GetBlockColumnStart(rel\_placement);

const auto& type = checked\_cast<const DateType&>(\*data->type());

switch (type.unit()) {

case DateUnit::DAY:

ConvertDatesShift<int32\_t, 1LL>(\*data, out\_values);

break;

case DateUnit::MILLI:

ConvertDatesShift<int64\_t, 86400000LL>(\*data, out\_values);

break;

}

return Status::OK();

}

protected:

Status Allocate() override {

RETURN\_NOT\_OK(this->AllocateNDArray(NPY\_DATETIME));

SetDatetimeUnit(NPY\_FR\_D);

return Status::OK();

}

};

template <TimeUnit::type UNIT>

class DatetimeWriter : public TypedPandasWriter<NPY\_DATETIME> {

public:

using TypedPandasWriter<NPY\_DATETIME>::TypedPandasWriter;

bool CanZeroCopy(const ChunkedArray& data) const override {

if (data.type()->id() == Type::TIMESTAMP) {

const auto& type = checked\_cast<const TimestampType&>(\*data.type());

return IsNonNullContiguous(data) && type.unit() == UNIT;

} else {

return false;

}

}

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

const auto& ts\_type = checked\_cast<const TimestampType&>(\*data->type());

DCHECK\_EQ(UNIT, ts\_type.unit()) << "Should only call instances of this writer "

<< "with arrays of the correct unit";

ConvertNumericNullable<int64\_t>(\*data, kPandasTimestampNull,

this->GetBlockColumnStart(rel\_placement));

return Status::OK();

}

protected:

Status Allocate() override {

RETURN\_NOT\_OK(this->AllocateNDArray(NPY\_DATETIME));

SetDatetimeUnit(internal::NumPyFrequency(UNIT));

return Status::OK();

}

};

using DatetimeSecondWriter = DatetimeWriter<TimeUnit::SECOND>;

class DatetimeMilliWriter : public DatetimeWriter<TimeUnit::MILLI> {

public:

using DatetimeWriter<TimeUnit::MILLI>::DatetimeWriter;

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

Type::type type = data->type()->id();

int64\_t\* out\_values = this->GetBlockColumnStart(rel\_placement);

if (type == Type::DATE32) {

// Convert from days since epoch to datetime64[ms]

ConvertDatetime<int32\_t, 86400000L>(\*data, out\_values);

} else if (type == Type::DATE64) {

ConvertNumericNullable<int64\_t>(\*data, kPandasTimestampNull, out\_values);

} else {

const auto& ts\_type = checked\_cast<const TimestampType&>(\*data->type());

DCHECK\_EQ(TimeUnit::MILLI, ts\_type.unit())

<< "Should only call instances of this writer "

<< "with arrays of the correct unit";

ConvertNumericNullable<int64\_t>(\*data, kPandasTimestampNull, out\_values);

}

return Status::OK();

}

};

using DatetimeMicroWriter = DatetimeWriter<TimeUnit::MICRO>;

class DatetimeNanoWriter : public DatetimeWriter<TimeUnit::NANO> {

public:

using DatetimeWriter<TimeUnit::NANO>::DatetimeWriter;

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

Type::type type = data->type()->id();

int64\_t\* out\_values = this->GetBlockColumnStart(rel\_placement);

compute::ExecContext ctx(options\_.pool);

compute::CastOptions options;

if (options\_.safe\_cast) {

options = compute::CastOptions::Safe();

} else {

options = compute::CastOptions::Unsafe();

}

Datum out;

auto target\_type = timestamp(TimeUnit::NANO);

if (type == Type::DATE32) {

// Convert from days since epoch to datetime64[ns]

ConvertDatetime<int32\_t, kNanosecondsInDay>(\*data, out\_values);

} else if (type == Type::DATE64) {

// Date64Type is millisecond timestamp stored as int64\_t

// TODO(wesm): Do we want to make sure to zero out the milliseconds?

ConvertDatetime<int64\_t, 1000000L>(\*data, out\_values);

} else if (type == Type::TIMESTAMP) {

const auto& ts\_type = checked\_cast<const TimestampType&>(\*data->type());

if (ts\_type.unit() == TimeUnit::NANO) {

ConvertNumericNullable<int64\_t>(\*data, kPandasTimestampNull, out\_values);

} else if (ts\_type.unit() == TimeUnit::MICRO || ts\_type.unit() == TimeUnit::MILLI ||

ts\_type.unit() == TimeUnit::SECOND) {

ARROW\_ASSIGN\_OR\_RAISE(out, compute::Cast(data, target\_type, options, &ctx));

ConvertNumericNullable<int64\_t>(\*out.chunked\_array(), kPandasTimestampNull,

out\_values);

} else {

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

}

} else {

return Status::NotImplemented("Cannot write Arrow data of type ",

data->type()->ToString(),

" to a Pandas datetime block.");

}

return Status::OK();

}

};

template <typename BASE>

class DatetimeTZWriter : public BASE {

public:

DatetimeTZWriter(const PandasOptions& options, const std::string& timezone,

int64\_t num\_rows)

: BASE(options, num\_rows, 1), timezone\_(timezone) {}

protected:

Status GetResultBlock(PyObject\*\* out) override {

RETURN\_NOT\_OK(this->MakeBlock1D());

\*out = this->block\_arr\_.obj();

return Status::OK();

}

Status AddResultMetadata(PyObject\* result) override {

PyObject\* py\_tz = PyUnicode\_FromStringAndSize(

timezone\_.c\_str(), static\_cast<Py\_ssize\_t>(timezone\_.size()));

RETURN\_IF\_PYERROR();

PyDict\_SetItemString(result, "timezone", py\_tz);

Py\_DECREF(py\_tz);

return Status::OK();

}

private:

std::string timezone\_;

};

using DatetimeSecondTZWriter = DatetimeTZWriter<DatetimeSecondWriter>;

using DatetimeMilliTZWriter = DatetimeTZWriter<DatetimeMilliWriter>;

using DatetimeMicroTZWriter = DatetimeTZWriter<DatetimeMicroWriter>;

using DatetimeNanoTZWriter = DatetimeTZWriter<DatetimeNanoWriter>;

template <TimeUnit::type UNIT>

class TimedeltaWriter : public TypedPandasWriter<NPY\_TIMEDELTA> {

public:

using TypedPandasWriter<NPY\_TIMEDELTA>::TypedPandasWriter;

Status AllocateTimedelta(int ndim) {

RETURN\_NOT\_OK(this->AllocateNDArray(NPY\_TIMEDELTA, ndim));

SetDatetimeUnit(internal::NumPyFrequency(UNIT));

return Status::OK();

}

bool CanZeroCopy(const ChunkedArray& data) const override {

const auto& type = checked\_cast<const DurationType&>(\*data.type());

return IsNonNullContiguous(data) && type.unit() == UNIT;

}

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

const auto& type = checked\_cast<const DurationType&>(\*data->type());

DCHECK\_EQ(UNIT, type.unit()) << "Should only call instances of this writer "

<< "with arrays of the correct unit";

ConvertNumericNullable<int64\_t>(\*data, kPandasTimestampNull,

this->GetBlockColumnStart(rel\_placement));

return Status::OK();

}

protected:

Status Allocate() override { return AllocateTimedelta(2); }

};

using TimedeltaSecondWriter = TimedeltaWriter<TimeUnit::SECOND>;

using TimedeltaMilliWriter = TimedeltaWriter<TimeUnit::MILLI>;

using TimedeltaMicroWriter = TimedeltaWriter<TimeUnit::MICRO>;

class TimedeltaNanoWriter : public TimedeltaWriter<TimeUnit::NANO> {

public:

using TimedeltaWriter<TimeUnit::NANO>::TimedeltaWriter;

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

Type::type type = data->type()->id();

int64\_t\* out\_values = this->GetBlockColumnStart(rel\_placement);

if (type == Type::DURATION) {

const auto& ts\_type = checked\_cast<const DurationType&>(\*data->type());

if (ts\_type.unit() == TimeUnit::NANO) {

ConvertNumericNullable<int64\_t>(\*data, kPandasTimestampNull, out\_values);

} else if (ts\_type.unit() == TimeUnit::MICRO) {

ConvertDatetime<int64\_t, 1000L>(\*data, out\_values);

} else if (ts\_type.unit() == TimeUnit::MILLI) {

ConvertDatetime<int64\_t, 1000000L>(\*data, out\_values);

} else if (ts\_type.unit() == TimeUnit::SECOND) {

ConvertDatetime<int64\_t, 1000000000L>(\*data, out\_values);

} else {

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

}

} else {

return Status::NotImplemented("Cannot write Arrow data of type ",

data->type()->ToString(),

" to a Pandas timedelta block.");

}

return Status::OK();

}

};

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

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

std::unique\_ptr<ArrayBuilder> builder;

RETURN\_NOT\_OK(MakeBuilder(default\_memory\_pool(), type, &builder));

RETURN\_NOT\_OK(builder->Resize(0));

return builder->Finish(out);

}

bool NeedDictionaryUnification(const ChunkedArray& data) {

if (data.num\_chunks() < 2) {

return false;

}

const auto& arr\_first = checked\_cast<const DictionaryArray&>(\*data.chunk(0));

for (int c = 1; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const DictionaryArray&>(\*data.chunk(c));

if (!(arr\_first.dictionary()->Equals(arr.dictionary()))) {

return true;

}

}

return false;

}

template <typename IndexType>

class CategoricalWriter

: public TypedPandasWriter<arrow\_traits<IndexType::type\_id>::npy\_type> {

public:

using TRAITS = arrow\_traits<IndexType::type\_id>;

using ArrayType = typename TypeTraits<IndexType>::ArrayType;

using T = typename TRAITS::T;

explicit CategoricalWriter(const PandasOptions& options, int64\_t num\_rows)

: TypedPandasWriter<TRAITS::npy\_type>(options, num\_rows, 1),

ordered\_(false),

needs\_copy\_(false) {}

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

return Status::NotImplemented("categorical type");

}

Status TransferSingle(std::shared\_ptr<ChunkedArray> data, PyObject\* py\_ref) override {

const auto& dict\_type = checked\_cast<const DictionaryType&>(\*data->type());

std::shared\_ptr<Array> dict;

if (data->num\_chunks() == 0) {

// no dictionary values => create empty array

RETURN\_NOT\_OK(this->AllocateNDArray(TRAITS::npy\_type, 1));

RETURN\_NOT\_OK(MakeZeroLengthArray(dict\_type.value\_type(), &dict));

} else {

DCHECK\_EQ(IndexType::type\_id, dict\_type.index\_type()->id());

RETURN\_NOT\_OK(WriteIndices(\*data, &dict));

}

PyObject\* pydict;

RETURN\_NOT\_OK(ConvertArrayToPandas(this->options\_, dict, nullptr, &pydict));

dictionary\_.reset(pydict);

ordered\_ = dict\_type.ordered();

return Status::OK();

}

Status Write(std::shared\_ptr<ChunkedArray> data, int64\_t abs\_placement,

int64\_t rel\_placement) override {

RETURN\_NOT\_OK(this->EnsurePlacementAllocated());

RETURN\_NOT\_OK(TransferSingle(data, /\*py\_ref=\*/nullptr));

this->placement\_data\_[rel\_placement] = abs\_placement;

return Status::OK();

}

Status GetSeriesResult(PyObject\*\* out) override {

PyAcquireGIL lock;

PyObject\* result = PyDict\_New();

RETURN\_IF\_PYERROR();

// Expected single array dictionary layout

PyDict\_SetItemString(result, "indices", this->block\_arr\_.obj());

RETURN\_IF\_PYERROR();

RETURN\_NOT\_OK(AddResultMetadata(result));

\*out = result;

return Status::OK();

}

protected:

Status AddResultMetadata(PyObject\* result) override {

PyDict\_SetItemString(result, "dictionary", dictionary\_.obj());

PyObject\* py\_ordered = ordered\_ ? Py\_True : Py\_False;

Py\_INCREF(py\_ordered);

PyDict\_SetItemString(result, "ordered", py\_ordered);

return Status::OK();

}

Status WriteIndicesUniform(const ChunkedArray& data) {

RETURN\_NOT\_OK(this->AllocateNDArray(TRAITS::npy\_type, 1));

T\* out\_values = reinterpret\_cast<T\*>(this->block\_data\_);

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const DictionaryArray&>(\*data.chunk(c));

const auto& indices = checked\_cast<const ArrayType&>(\*arr.indices());

auto values = reinterpret\_cast<const T\*>(indices.raw\_values());

RETURN\_NOT\_OK(CheckIndexBounds(\*indices.data(), arr.dictionary()->length()));

// Null is -1 in CategoricalBlock

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

if (indices.IsValid(i)) {

\*out\_values++ = values[i];

} else {

\*out\_values++ = -1;

}

}

}

return Status::OK();

}

Status WriteIndicesVarying(const ChunkedArray& data, std::shared\_ptr<Array>\* out\_dict) {

// Yield int32 indices to allow for dictionary outgrowing the current index

// type

RETURN\_NOT\_OK(this->AllocateNDArray(NPY\_INT32, 1));

auto out\_values = reinterpret\_cast<int32\_t\*>(this->block\_data\_);

const auto& dict\_type = checked\_cast<const DictionaryType&>(\*data.type());

ARROW\_ASSIGN\_OR\_RAISE(auto unifier, DictionaryUnifier::Make(dict\_type.value\_type(),

this->options\_.pool));

for (int c = 0; c < data.num\_chunks(); c++) {

const auto& arr = checked\_cast<const DictionaryArray&>(\*data.chunk(c));

const auto& indices = checked\_cast<const ArrayType&>(\*arr.indices());

auto values = reinterpret\_cast<const T\*>(indices.raw\_values());

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

RETURN\_NOT\_OK(unifier->Unify(\*arr.dictionary(), &transpose\_buffer));

auto transpose = reinterpret\_cast<const int32\_t\*>(transpose\_buffer->data());

int64\_t dict\_length = arr.dictionary()->length();

RETURN\_NOT\_OK(CheckIndexBounds(\*indices.data(), dict\_length));

// Null is -1 in CategoricalBlock

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

if (indices.IsValid(i)) {

\*out\_values++ = transpose[values[i]];

} else {

\*out\_values++ = -1;

}

}

}

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

return unifier->GetResult(&unused\_type, out\_dict);

}

Status WriteIndices(const ChunkedArray& data, std::shared\_ptr<Array>\* out\_dict) {

DCHECK\_GT(data.num\_chunks(), 0);

// Sniff the first chunk

const auto& arr\_first = checked\_cast<const DictionaryArray&>(\*data.chunk(0));

const auto indices\_first = std::static\_pointer\_cast<ArrayType>(arr\_first.indices());

if (data.num\_chunks() == 1 && indices\_first->null\_count() == 0) {

RETURN\_NOT\_OK(

CheckIndexBounds(\*indices\_first->data(), arr\_first.dictionary()->length()));

PyObject\* wrapped;

npy\_intp dims[1] = {static\_cast<npy\_intp>(this->num\_rows\_)};

RETURN\_NOT\_OK(MakeNumPyView(indices\_first, /\*py\_ref=\*/nullptr, TRAITS::npy\_type,

/\*ndim=\*/1, dims, &wrapped));

this->SetBlockData(wrapped);

\*out\_dict = arr\_first.dictionary();

} else {

RETURN\_NOT\_OK(this->CheckNotZeroCopyOnly(data));

if (NeedDictionaryUnification(data)) {

RETURN\_NOT\_OK(WriteIndicesVarying(data, out\_dict));

} else {

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

\*out\_dict = arr\_first.dictionary();

}

}

return Status::OK();

}

OwnedRefNoGIL dictionary\_;

bool ordered\_;

bool needs\_copy\_;

};

class ExtensionWriter : public PandasWriter {

public:

using PandasWriter::PandasWriter;

Status Allocate() override {

// no-op

return Status::OK();

}

Status TransferSingle(std::shared\_ptr<ChunkedArray> data, PyObject\* py\_ref) override {

PyAcquireGIL lock;

PyObject\* py\_array;

py\_array = wrap\_chunked\_array(data);

py\_array\_.reset(py\_array);

return Status::OK();

}

Status CopyInto(std::shared\_ptr<ChunkedArray> data, int64\_t rel\_placement) override {

return TransferSingle(data, nullptr);

}

Status GetDataFrameResult(PyObject\*\* out) override {

PyAcquireGIL lock;

PyObject\* result = PyDict\_New();

RETURN\_IF\_PYERROR();

PyDict\_SetItemString(result, "py\_array", py\_array\_.obj());

PyDict\_SetItemString(result, "placement", placement\_arr\_.obj());

\*out = result;

return Status::OK();

}

Status GetSeriesResult(PyObject\*\* out) override {

\*out = py\_array\_.detach();

return Status::OK();

}

protected:

OwnedRefNoGIL py\_array\_;

};

Status MakeWriter(const PandasOptions& options, PandasWriter::type writer\_type,

const DataType& type, int64\_t num\_rows, int num\_columns,

std::shared\_ptr<PandasWriter>\* writer) {

#define BLOCK\_CASE(NAME, TYPE) \

case PandasWriter::NAME: \

\*writer = std::make\_shared<TYPE>(options, num\_rows, num\_columns); \

break;

#define CATEGORICAL\_CASE(TYPE) \

case TYPE::type\_id: \

\*writer = std::make\_shared<CategoricalWriter<TYPE>>(options, num\_rows); \

break;

#define TZ\_CASE(NAME, TYPE) \

case PandasWriter::NAME: { \

const auto& ts\_type = checked\_cast<const TimestampType&>(type); \

\*writer = std::make\_shared<TYPE>(options, ts\_type.timezone(), num\_rows); \

} break;

switch (writer\_type) {

case PandasWriter::CATEGORICAL: {

const auto& index\_type = \*checked\_cast<const DictionaryType&>(type).index\_type();

switch (index\_type.id()) {

CATEGORICAL\_CASE(Int8Type);

CATEGORICAL\_CASE(Int16Type);

CATEGORICAL\_CASE(Int32Type);

CATEGORICAL\_CASE(Int64Type);

case Type::UINT8:

case Type::UINT16:

case Type::UINT32:

case Type::UINT64:

return Status::TypeError(

"Converting unsigned dictionary indices to pandas",

" not yet supported, index type: ", index\_type.ToString());

default:

// Unreachable

DCHECK(false);

break;

}

} break;

case PandasWriter::EXTENSION:

\*writer = std::make\_shared<ExtensionWriter>(options, num\_rows, num\_columns);

break;

BLOCK\_CASE(OBJECT, ObjectWriter);

BLOCK\_CASE(UINT8, UInt8Writer);

BLOCK\_CASE(INT8, Int8Writer);

BLOCK\_CASE(UINT16, UInt16Writer);

BLOCK\_CASE(INT16, Int16Writer);

BLOCK\_CASE(UINT32, UInt32Writer);

BLOCK\_CASE(INT32, Int32Writer);

BLOCK\_CASE(UINT64, UInt64Writer);

BLOCK\_CASE(INT64, Int64Writer);

BLOCK\_CASE(HALF\_FLOAT, Float16Writer);

BLOCK\_CASE(FLOAT, Float32Writer);

BLOCK\_CASE(DOUBLE, Float64Writer);

BLOCK\_CASE(BOOL, BoolWriter);

BLOCK\_CASE(DATETIME\_DAY, DatetimeDayWriter);

BLOCK\_CASE(DATETIME\_SECOND, DatetimeSecondWriter);

BLOCK\_CASE(DATETIME\_MILLI, DatetimeMilliWriter);

BLOCK\_CASE(DATETIME\_MICRO, DatetimeMicroWriter);

BLOCK\_CASE(DATETIME\_NANO, DatetimeNanoWriter);

BLOCK\_CASE(TIMEDELTA\_SECOND, TimedeltaSecondWriter);

BLOCK\_CASE(TIMEDELTA\_MILLI, TimedeltaMilliWriter);

BLOCK\_CASE(TIMEDELTA\_MICRO, TimedeltaMicroWriter);

BLOCK\_CASE(TIMEDELTA\_NANO, TimedeltaNanoWriter);

TZ\_CASE(DATETIME\_SECOND\_TZ, DatetimeSecondTZWriter);

TZ\_CASE(DATETIME\_MILLI\_TZ, DatetimeMilliTZWriter);

TZ\_CASE(DATETIME\_MICRO\_TZ, DatetimeMicroTZWriter);

TZ\_CASE(DATETIME\_NANO\_TZ, DatetimeNanoTZWriter);

default:

return Status::NotImplemented("Unsupported block type");

}

#undef BLOCK\_CASE

#undef CATEGORICAL\_CASE

return Status::OK();

}

static Status GetPandasWriterType(const ChunkedArray& data, const PandasOptions& options,

PandasWriter::type\* output\_type) {

#define INTEGER\_CASE(NAME) \

\*output\_type = \

data.null\_count() > 0 \

? options.integer\_object\_nulls ? PandasWriter::OBJECT : PandasWriter::DOUBLE \

: PandasWriter::NAME; \

break;

switch (data.type()->id()) {

case Type::BOOL:

\*output\_type = data.null\_count() > 0 ? PandasWriter::OBJECT : PandasWriter::BOOL;

break;

case Type::UINT8:

INTEGER\_CASE(UINT8);

case Type::INT8:

INTEGER\_CASE(INT8);

case Type::UINT16:

INTEGER\_CASE(UINT16);

case Type::INT16:

INTEGER\_CASE(INT16);

case Type::UINT32:

INTEGER\_CASE(UINT32);

case Type::INT32:

INTEGER\_CASE(INT32);

case Type::UINT64:

INTEGER\_CASE(UINT64);

case Type::INT64:

INTEGER\_CASE(INT64);

case Type::HALF\_FLOAT:

\*output\_type = PandasWriter::HALF\_FLOAT;

break;

case Type::FLOAT:

\*output\_type = PandasWriter::FLOAT;

break;

case Type::DOUBLE:

\*output\_type = PandasWriter::DOUBLE;

break;

case Type::STRING: // fall through

case Type::LARGE\_STRING: // fall through

case Type::STRING\_VIEW: // fall through

case Type::BINARY: // fall through

case Type::LARGE\_BINARY:

case Type::BINARY\_VIEW:

case Type::NA: // fall through

case Type::FIXED\_SIZE\_BINARY: // fall through

case Type::STRUCT: // fall through

case Type::TIME32: // fall through

case Type::TIME64: // fall through

case Type::DECIMAL128: // fall through

case Type::DECIMAL256: // fall through

case Type::INTERVAL\_MONTH\_DAY\_NANO: // fall through

\*output\_type = PandasWriter::OBJECT;

break;

case Type::DATE32:

if (options.date\_as\_object) {

\*output\_type = PandasWriter::OBJECT;

} else if (options.coerce\_temporal\_nanoseconds) {

\*output\_type = PandasWriter::DATETIME\_NANO;

} else if (options.to\_numpy) {

// Numpy supports Day, but Pandas does not

\*output\_type = PandasWriter::DATETIME\_DAY;

} else {

\*output\_type = PandasWriter::DATETIME\_MILLI;

}

break;

case Type::DATE64:

if (options.date\_as\_object) {

\*output\_type = PandasWriter::OBJECT;

} else if (options.coerce\_temporal\_nanoseconds) {

\*output\_type = PandasWriter::DATETIME\_NANO;

} else {

\*output\_type = PandasWriter::DATETIME\_MILLI;

}

break;

case Type::TIMESTAMP: {

const auto& ts\_type = checked\_cast<const TimestampType&>(\*data.type());

if (options.timestamp\_as\_object && ts\_type.unit() != TimeUnit::NANO) {

// Nanoseconds are never out of bounds for pandas, so in that case

// we don't convert to object

\*output\_type = PandasWriter::OBJECT;

} else if (options.coerce\_temporal\_nanoseconds) {

if (!ts\_type.timezone().empty()) {

\*output\_type = PandasWriter::DATETIME\_NANO\_TZ;

} else {

\*output\_type = PandasWriter::DATETIME\_NANO;

}

} else {

if (!ts\_type.timezone().empty()) {

switch (ts\_type.unit()) {

case TimeUnit::SECOND:

\*output\_type = PandasWriter::DATETIME\_SECOND\_TZ;

break;

case TimeUnit::MILLI:

\*output\_type = PandasWriter::DATETIME\_MILLI\_TZ;

break;

case TimeUnit::MICRO:

\*output\_type = PandasWriter::DATETIME\_MICRO\_TZ;

break;

case TimeUnit::NANO:

\*output\_type = PandasWriter::DATETIME\_NANO\_TZ;

break;

}

} else {

switch (ts\_type.unit()) {

case TimeUnit::SECOND:

\*output\_type = PandasWriter::DATETIME\_SECOND;

break;

case TimeUnit::MILLI:

\*output\_type = PandasWriter::DATETIME\_MILLI;

break;

case TimeUnit::MICRO:

\*output\_type = PandasWriter::DATETIME\_MICRO;

break;

case TimeUnit::NANO:

\*output\_type = PandasWriter::DATETIME\_NANO;

break;

}

}

}

} break;

case Type::DURATION: {

const auto& dur\_type = checked\_cast<const DurationType&>(\*data.type());

if (options.coerce\_temporal\_nanoseconds) {

\*output\_type = PandasWriter::TIMEDELTA\_NANO;

} else {

switch (dur\_type.unit()) {

case TimeUnit::SECOND:

\*output\_type = PandasWriter::TIMEDELTA\_SECOND;

break;

case TimeUnit::MILLI:

\*output\_type = PandasWriter::TIMEDELTA\_MILLI;

break;

case TimeUnit::MICRO:

\*output\_type = PandasWriter::TIMEDELTA\_MICRO;

break;

case TimeUnit::NANO:

\*output\_type = PandasWriter::TIMEDELTA\_NANO;

break;

}

}

} break;

case Type::FIXED\_SIZE\_LIST:

case Type::LIST:

case Type::LARGE\_LIST:

case Type::LIST\_VIEW:

case Type::LARGE\_LIST\_VIEW:

case Type::MAP: {

auto list\_type = std::static\_pointer\_cast<BaseListType>(data.type());

if (!ListTypeSupported(\*list\_type->value\_type())) {

return Status::NotImplemented("Not implemented type for Arrow list to pandas: ",

list\_type->value\_type()->ToString());

}

\*output\_type = PandasWriter::OBJECT;

} break;

case Type::DICTIONARY:

\*output\_type = PandasWriter::CATEGORICAL;

break;

case Type::EXTENSION:

\*output\_type = PandasWriter::EXTENSION;

break;

default:

return Status::NotImplemented(

"No known equivalent Pandas block for Arrow data of type ",

data.type()->ToString(), " is known.");

}

return Status::OK();

}

// Construct the exact pandas "BlockManager" memory layout

//

// \* For each column determine the correct output pandas type

// \* Allocate 2D blocks (ncols x nrows) for each distinct data type in output

// \* Allocate block placement arrays

// \* Write Arrow columns out into each slice of memory; populate block

// \* placement arrays as we go

class PandasBlockCreator {

public:

using WriterMap = std::unordered\_map<int, std::shared\_ptr<PandasWriter>>;

explicit PandasBlockCreator(const PandasOptions& options, FieldVector fields,

ChunkedArrayVector arrays)

: options\_(options), fields\_(std::move(fields)), arrays\_(std::move(arrays)) {

num\_columns\_ = static\_cast<int>(arrays\_.size());

if (num\_columns\_ > 0) {

num\_rows\_ = arrays\_[0]->length();

}

column\_block\_placement\_.resize(num\_columns\_);

}

virtual ~PandasBlockCreator() = default;

virtual Status Convert(PyObject\*\* out) = 0;

Status AppendBlocks(const WriterMap& blocks, PyObject\* list) {

for (const auto& it : blocks) {

PyObject\* item;

RETURN\_NOT\_OK(it.second->GetDataFrameResult(&item));

if (PyList\_Append(list, item) < 0) {

RETURN\_IF\_PYERROR();

}

// ARROW-1017; PyList\_Append increments object refcount

Py\_DECREF(item);

}

return Status::OK();

}

protected:

PandasOptions options\_;

FieldVector fields\_;

ChunkedArrayVector arrays\_;

int num\_columns\_;

int64\_t num\_rows\_;

// column num -> relative placement within internal block

std::vector<int> column\_block\_placement\_;

};

// Helper function for extension chunked arrays

// Constructing a storage chunked array of an extension chunked array

std::shared\_ptr<ChunkedArray> GetStorageChunkedArray(std::shared\_ptr<ChunkedArray> arr) {

auto value\_type = checked\_cast<const ExtensionType&>(\*arr->type()).storage\_type();

ArrayVector storage\_arrays;

for (int c = 0; c < arr->num\_chunks(); c++) {

const auto& arr\_ext = checked\_cast<const ExtensionArray&>(\*arr->chunk(c));

storage\_arrays.emplace\_back(arr\_ext.storage());

}

return std::make\_shared<ChunkedArray>(std::move(storage\_arrays), value\_type);

};

// Helper function to decode RunEndEncodedArray

Result<std::shared\_ptr<ChunkedArray>> GetDecodedChunkedArray(

std::shared\_ptr<ChunkedArray> arr) {

ARROW\_ASSIGN\_OR\_RAISE(Datum decoded, compute::RunEndDecode(arr));

DCHECK(decoded.is\_chunked\_array());

return decoded.chunked\_array();

};

class ConsolidatedBlockCreator : public PandasBlockCreator {

public:

using PandasBlockCreator::PandasBlockCreator;

Status Convert(PyObject\*\* out) override {

column\_types\_.resize(num\_columns\_);

RETURN\_NOT\_OK(CreateBlocks());

RETURN\_NOT\_OK(WriteTableToBlocks());

PyAcquireGIL lock;

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

RETURN\_IF\_PYERROR();

RETURN\_NOT\_OK(AppendBlocks(blocks\_, result));

RETURN\_NOT\_OK(AppendBlocks(singleton\_blocks\_, result));

\*out = result;

return Status::OK();

}

Status GetBlockType(int column\_index, PandasWriter::type\* out) {

if (options\_.extension\_columns.count(fields\_[column\_index]->name())) {

\*out = PandasWriter::EXTENSION;

return Status::OK();

} else {

// In case of an extension array default to the storage type

if (arrays\_[column\_index]->type()->id() == Type::EXTENSION) {

arrays\_[column\_index] = GetStorageChunkedArray(arrays\_[column\_index]);

}

// In case of a RunEndEncodedArray default to the values type

else if (arrays\_[column\_index]->type()->id() == Type::RUN\_END\_ENCODED) {

ARROW\_ASSIGN\_OR\_RAISE(arrays\_[column\_index],

GetDecodedChunkedArray(arrays\_[column\_index]));

}

return GetPandasWriterType(\*arrays\_[column\_index], options\_, out);

}

}

Status CreateBlocks() {

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

const DataType& type = \*arrays\_[i]->type();

PandasWriter::type output\_type;

RETURN\_NOT\_OK(GetBlockType(i, &output\_type));

int block\_placement = 0;

std::shared\_ptr<PandasWriter> writer;

if (output\_type == PandasWriter::CATEGORICAL ||

output\_type == PandasWriter::DATETIME\_SECOND\_TZ ||

output\_type == PandasWriter::DATETIME\_MILLI\_TZ ||

output\_type == PandasWriter::DATETIME\_MICRO\_TZ ||

output\_type == PandasWriter::DATETIME\_NANO\_TZ ||

output\_type == PandasWriter::EXTENSION) {

RETURN\_NOT\_OK(MakeWriter(options\_, output\_type, type, num\_rows\_,

/\*num\_columns=\*/1, &writer));

singleton\_blocks\_[i] = writer;

} else {

auto it = block\_sizes\_.find(output\_type);

if (it != block\_sizes\_.end()) {

block\_placement = it->second;

// Increment count

++it->second;

} else {

// Add key to map

block\_sizes\_[output\_type] = 1;

}

}

column\_types\_[i] = output\_type;

column\_block\_placement\_[i] = block\_placement;

}

// Create normal non-categorical blocks

for (const auto& it : this->block\_sizes\_) {

PandasWriter::type output\_type = static\_cast<PandasWriter::type>(it.first);

std::shared\_ptr<PandasWriter> block;

RETURN\_NOT\_OK(MakeWriter(this->options\_, output\_type, /\*unused\*/ \*null(), num\_rows\_,

it.second, &block));

this->blocks\_[output\_type] = block;

}

return Status::OK();

}

Status GetWriter(int i, std::shared\_ptr<PandasWriter>\* block) {

PandasWriter::type output\_type = this->column\_types\_[i];

switch (output\_type) {

case PandasWriter::CATEGORICAL:

case PandasWriter::DATETIME\_SECOND\_TZ:

case PandasWriter::DATETIME\_MILLI\_TZ:

case PandasWriter::DATETIME\_MICRO\_TZ:

case PandasWriter::DATETIME\_NANO\_TZ:

case PandasWriter::EXTENSION: {

auto it = this->singleton\_blocks\_.find(i);

if (it == this->singleton\_blocks\_.end()) {

return Status::KeyError("No block allocated");

}

\*block = it->second;

} break;

default:

auto it = this->blocks\_.find(output\_type);

if (it == this->blocks\_.end()) {

return Status::KeyError("No block allocated");

}

\*block = it->second;

break;

}

return Status::OK();

}

Status WriteTableToBlocks() {

auto WriteColumn = [this](int i) {

std::shared\_ptr<PandasWriter> block;

RETURN\_NOT\_OK(this->GetWriter(i, &block));

// ARROW-3789 Use std::move on the array to permit self-destructing

return block->Write(std::move(arrays\_[i]), i, this->column\_block\_placement\_[i]);

};

return OptionalParallelFor(options\_.use\_threads, num\_columns\_, WriteColumn);

}

private:

// column num -> block type id

std::vector<PandasWriter::type> column\_types\_;

// block type -> type count

std::unordered\_map<int, int> block\_sizes\_;

std::unordered\_map<int, const DataType\*> block\_types\_;

// block type -> block

WriterMap blocks\_;

WriterMap singleton\_blocks\_;

};

/// \brief Create blocks for pandas.DataFrame block manager using one block per

/// column strategy. This permits some zero-copy optimizations as well as the

/// ability for the table to "self-destruct" if selected by the user.

class SplitBlockCreator : public PandasBlockCreator {

public:

using PandasBlockCreator::PandasBlockCreator;

Status GetWriter(int i, std::shared\_ptr<PandasWriter>\* writer) {

PandasWriter::type output\_type = PandasWriter::OBJECT;

const DataType& type = \*arrays\_[i]->type();

if (options\_.extension\_columns.count(fields\_[i]->name())) {

output\_type = PandasWriter::EXTENSION;

} else {

// Null count needed to determine output type

RETURN\_NOT\_OK(GetPandasWriterType(\*arrays\_[i], options\_, &output\_type));

}

return MakeWriter(this->options\_, output\_type, type, num\_rows\_, 1, writer);

}

Status Convert(PyObject\*\* out) override {

PyAcquireGIL lock;

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

RETURN\_IF\_PYERROR();

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

std::shared\_ptr<PandasWriter> writer;

RETURN\_NOT\_OK(GetWriter(i, &writer));

// ARROW-3789 Use std::move on the array to permit self-destructing

RETURN\_NOT\_OK(writer->Write(std::move(arrays\_[i]), i, /\*rel\_placement=\*/0));

PyObject\* item;

RETURN\_NOT\_OK(writer->GetDataFrameResult(&item));

if (PyList\_Append(result, item) < 0) {

RETURN\_IF\_PYERROR();

}

// PyList\_Append increments object refcount

Py\_DECREF(item);

}

\*out = result;

return Status::OK();

}

private:

std::vector<std::shared\_ptr<PandasWriter>> writers\_;

};

Status ConvertCategoricals(const PandasOptions& options, ChunkedArrayVector\* arrays,

FieldVector\* fields) {

std::vector<int> columns\_to\_encode;

// For Categorical conversions

auto EncodeColumn = [&](int j) {

int i = columns\_to\_encode[j];

if (options.zero\_copy\_only) {

return Status::Invalid("Need to dictionary encode a column, but ",

"only zero-copy conversions allowed");

}

compute::ExecContext ctx(options.pool);

ARROW\_ASSIGN\_OR\_RAISE(

Datum out, DictionaryEncode((\*arrays)[i],

compute::DictionaryEncodeOptions::Defaults(), &ctx));

(\*arrays)[i] = out.chunked\_array();

(\*fields)[i] = (\*fields)[i]->WithType((\*arrays)[i]->type());

return Status::OK();

};

if (!options.categorical\_columns.empty()) {

for (int i = 0; i < static\_cast<int>(arrays->size()); i++) {

if ((\*arrays)[i]->type()->id() != Type::DICTIONARY &&

options.categorical\_columns.count((\*fields)[i]->name())) {

columns\_to\_encode.push\_back(i);

}

}

}

if (options.strings\_to\_categorical) {

for (int i = 0; i < static\_cast<int>(arrays->size()); i++) {

if (is\_base\_binary\_like((\*arrays)[i]->type()->id())) {

columns\_to\_encode.push\_back(i);

}

}

}

return OptionalParallelFor(options.use\_threads,

static\_cast<int>(columns\_to\_encode.size()), EncodeColumn);

}

} // namespace

Status ConvertArrayToPandas(const PandasOptions& options, std::shared\_ptr<Array> arr,

PyObject\* py\_ref, PyObject\*\* out) {

return ConvertChunkedArrayToPandas(

options, std::make\_shared<ChunkedArray>(std::move(arr)), py\_ref, out);

}

Status ConvertChunkedArrayToPandas(const PandasOptions& options,

std::shared\_ptr<ChunkedArray> arr, PyObject\* py\_ref,

PyObject\*\* out) {

if (options.decode\_dictionaries && arr->type()->id() == Type::DICTIONARY) {

// XXX we should return an error as below if options.zero\_copy\_only

// is true, but that would break compatibility with existing tests.

const auto& dense\_type =

checked\_cast<const DictionaryType&>(\*arr->type()).value\_type();

RETURN\_NOT\_OK(DecodeDictionaries(options.pool, dense\_type, &arr));

DCHECK\_NE(arr->type()->id(), Type::DICTIONARY);

// The original Python DictionaryArray won't own the memory anymore

// as we actually built a new array when we decoded the DictionaryArray

// thus let the final resulting numpy array own the memory through a Capsule

py\_ref = nullptr;

}

if (options.strings\_to\_categorical && is\_base\_binary\_like(arr->type()->id())) {

if (options.zero\_copy\_only) {

return Status::Invalid("Need to dictionary encode a column, but ",

"only zero-copy conversions allowed");

}

compute::ExecContext ctx(options.pool);

ARROW\_ASSIGN\_OR\_RAISE(

Datum out,

DictionaryEncode(arr, compute::DictionaryEncodeOptions::Defaults(), &ctx));

arr = out.chunked\_array();

}

PandasOptions modified\_options = options;

modified\_options.strings\_to\_categorical = false;

// ARROW-7596: We permit the hybrid Series/DataFrame code path to do zero copy

// optimizations that we do not allow in the default case when converting

// Table->DataFrame

modified\_options.allow\_zero\_copy\_blocks = true;

// In case of an extension array default to the storage type

if (arr->type()->id() == Type::EXTENSION) {

arr = GetStorageChunkedArray(arr);

}

// In case of a RunEndEncodedArray decode the array

else if (arr->type()->id() == Type::RUN\_END\_ENCODED) {

if (options.zero\_copy\_only) {

return Status::Invalid("Need to dencode a RunEndEncodedArray, but ",

"only zero-copy conversions allowed");

}

ARROW\_ASSIGN\_OR\_RAISE(arr, GetDecodedChunkedArray(arr));

// Because we built a new array when we decoded the RunEndEncodedArray

// the final resulting numpy array should own the memory through a Capsule

py\_ref = nullptr;

}

PandasWriter::type output\_type;

RETURN\_NOT\_OK(GetPandasWriterType(\*arr, modified\_options, &output\_type));

if (options.decode\_dictionaries) {

DCHECK\_NE(output\_type, PandasWriter::CATEGORICAL);

}

std::shared\_ptr<PandasWriter> writer;

RETURN\_NOT\_OK(MakeWriter(modified\_options, output\_type, \*arr->type(), arr->length(),

/\*num\_columns=\*/1, &writer));

RETURN\_NOT\_OK(writer->TransferSingle(std::move(arr), py\_ref));

return writer->GetSeriesResult(out);

}

Status ConvertTableToPandas(const PandasOptions& options, std::shared\_ptr<Table> table,

PyObject\*\* out) {

ChunkedArrayVector arrays = table->columns();

FieldVector fields = table->fields();

// ARROW-3789: allow "self-destructing" by releasing references to columns as

// we convert them to pandas

table = nullptr;

RETURN\_NOT\_OK(ConvertCategoricals(options, &arrays, &fields));

PandasOptions modified\_options = options;

modified\_options.strings\_to\_categorical = false;

modified\_options.categorical\_columns.clear();

if (options.split\_blocks) {

modified\_options.allow\_zero\_copy\_blocks = true;

SplitBlockCreator helper(modified\_options, std::move(fields), std::move(arrays));

return helper.Convert(out);

} else {

ConsolidatedBlockCreator helper(modified\_options, std::move(fields),

std::move(arrays));

return helper.Convert(out);

}

}

} // namespace py

} // namespace arrow