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

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

// distributed with this work for additional information

// regarding copyright ownership. The ASF licenses this file

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

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

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

//

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

//

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

// software distributed under the License is distributed on an

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

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

// specific language governing permissions and limitations

// under the License.

#include "arrow/python/inference.h"

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

#include <datetime.h>

#include <algorithm>

#include <limits>

#include <map>

#include <string>

#include <utility>

#include <vector>

#include "arrow/scalar.h"

#include "arrow/status.h"

#include "arrow/util/decimal.h"

#include "arrow/util/logging.h"

#include "arrow/python/datetime.h"

#include "arrow/python/decimal.h"

#include "arrow/python/helpers.h"

#include "arrow/python/iterators.h"

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

namespace arrow {

namespace py {

namespace {

// Assigns a tuple to interval\_types\_tuple containing the nametuple for

// MonthDayNanoIntervalType and if present dateutil's relativedelta and

// pandas DateOffset.

Status ImportPresentIntervalTypes(OwnedRefNoGIL\* interval\_types\_tuple) {

OwnedRef relative\_delta\_module;

// These are Optional imports so swallow errors.

OwnedRef relative\_delta\_type;

// Try to import pandas to get types.

internal::InitPandasStaticData();

if (internal::ImportModule("dateutil.relativedelta", &relative\_delta\_module).ok()) {

RETURN\_NOT\_OK(internal::ImportFromModule(relative\_delta\_module.obj(), "relativedelta",

&relative\_delta\_type));

}

PyObject\* date\_offset\_type = internal::BorrowPandasDataOffsetType();

interval\_types\_tuple->reset(

PyTuple\_New(1 + (date\_offset\_type != nullptr ? 1 : 0) +

(relative\_delta\_type.obj() != nullptr ? 1 : 0)));

RETURN\_IF\_PYERROR();

int index = 0;

PyTuple\_SetItem(interval\_types\_tuple->obj(), index++,

internal::NewMonthDayNanoTupleType());

RETURN\_IF\_PYERROR();

if (date\_offset\_type != nullptr) {

Py\_XINCREF(date\_offset\_type);

PyTuple\_SetItem(interval\_types\_tuple->obj(), index++, date\_offset\_type);

RETURN\_IF\_PYERROR();

}

if (relative\_delta\_type.obj() != nullptr) {

PyTuple\_SetItem(interval\_types\_tuple->obj(), index++, relative\_delta\_type.detach());

RETURN\_IF\_PYERROR();

}

return Status::OK();

}

} // namespace

#define \_NUMPY\_UNIFY\_NOOP(DTYPE) \

case NPY\_##DTYPE: \

return OK;

#define \_NUMPY\_UNIFY\_PROMOTE(DTYPE) \

case NPY\_##DTYPE: \

current\_type\_num\_ = dtype; \

current\_dtype\_ = descr; \

return OK;

#define \_NUMPY\_UNIFY\_PROMOTE\_TO(DTYPE, NEW\_TYPE) \

case NPY\_##DTYPE: \

current\_type\_num\_ = NPY\_##NEW\_TYPE; \

current\_dtype\_ = PyArray\_DescrFromType(current\_type\_num\_); \

return OK;

// Form a consensus NumPy dtype to use for Arrow conversion for a

// collection of dtype objects observed one at a time

class NumPyDtypeUnifier {

public:

enum Action { OK, INVALID };

NumPyDtypeUnifier() : current\_type\_num\_(-1), current\_dtype\_(nullptr) {}

Status InvalidMix(int new\_dtype) {

return Status::Invalid("Cannot mix NumPy dtypes ",

GetNumPyTypeName(current\_type\_num\_), " and ",

GetNumPyTypeName(new\_dtype));

}

int Observe\_BOOL(PyArray\_Descr\* descr, int dtype) { return INVALID; }

int Observe\_INT8(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_PROMOTE(INT16);

\_NUMPY\_UNIFY\_PROMOTE(INT32);

\_NUMPY\_UNIFY\_PROMOTE(INT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT32);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_INT16(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(INT8);

\_NUMPY\_UNIFY\_PROMOTE(INT32);

\_NUMPY\_UNIFY\_PROMOTE(INT64);

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT32);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_INT32(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(INT8);

\_NUMPY\_UNIFY\_NOOP(INT16);

\_NUMPY\_UNIFY\_PROMOTE(INT32);

\_NUMPY\_UNIFY\_PROMOTE(INT64);

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_NOOP(UINT16);

\_NUMPY\_UNIFY\_PROMOTE\_TO(FLOAT32, FLOAT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_INT64(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(INT8);

\_NUMPY\_UNIFY\_NOOP(INT16);

\_NUMPY\_UNIFY\_NOOP(INT32);

\_NUMPY\_UNIFY\_NOOP(INT64);

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_NOOP(UINT16);

\_NUMPY\_UNIFY\_NOOP(UINT32);

\_NUMPY\_UNIFY\_PROMOTE\_TO(FLOAT32, FLOAT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_UINT8(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_PROMOTE(UINT16);

\_NUMPY\_UNIFY\_PROMOTE(UINT32);

\_NUMPY\_UNIFY\_PROMOTE(UINT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT32);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_UINT16(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_PROMOTE(UINT32);

\_NUMPY\_UNIFY\_PROMOTE(UINT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT32);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_UINT32(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_NOOP(UINT16);

\_NUMPY\_UNIFY\_PROMOTE(UINT64);

\_NUMPY\_UNIFY\_PROMOTE\_TO(FLOAT32, FLOAT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_UINT64(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_NOOP(UINT16);

\_NUMPY\_UNIFY\_NOOP(UINT32);

\_NUMPY\_UNIFY\_PROMOTE\_TO(FLOAT32, FLOAT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_FLOAT16(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_PROMOTE(FLOAT32);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_FLOAT32(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(INT8);

\_NUMPY\_UNIFY\_NOOP(INT16);

\_NUMPY\_UNIFY\_NOOP(INT32);

\_NUMPY\_UNIFY\_NOOP(INT64);

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_NOOP(UINT16);

\_NUMPY\_UNIFY\_NOOP(UINT32);

\_NUMPY\_UNIFY\_NOOP(UINT64);

\_NUMPY\_UNIFY\_PROMOTE(FLOAT64);

default:

return INVALID;

}

}

int Observe\_FLOAT64(PyArray\_Descr\* descr, int dtype) {

switch (dtype) {

\_NUMPY\_UNIFY\_NOOP(INT8);

\_NUMPY\_UNIFY\_NOOP(INT16);

\_NUMPY\_UNIFY\_NOOP(INT32);

\_NUMPY\_UNIFY\_NOOP(INT64);

\_NUMPY\_UNIFY\_NOOP(UINT8);

\_NUMPY\_UNIFY\_NOOP(UINT16);

\_NUMPY\_UNIFY\_NOOP(UINT32);

\_NUMPY\_UNIFY\_NOOP(UINT64);

default:

return INVALID;

}

}

int Observe\_DATETIME(PyArray\_Descr\* dtype\_obj) {

// TODO: check that units are all the same

return OK;

}

Status Observe(PyArray\_Descr\* descr) {

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

if (current\_type\_num\_ == -1) {

current\_dtype\_ = descr;

current\_type\_num\_ = dtype;

return Status::OK();

} else if (current\_type\_num\_ == dtype) {

return Status::OK();

}

#define OBSERVE\_CASE(DTYPE) \

case NPY\_##DTYPE: \

action = Observe\_##DTYPE(descr, dtype); \

break;

int action = OK;

switch (current\_type\_num\_) {

OBSERVE\_CASE(BOOL);

OBSERVE\_CASE(INT8);

OBSERVE\_CASE(INT16);

OBSERVE\_CASE(INT32);

OBSERVE\_CASE(INT64);

OBSERVE\_CASE(UINT8);

OBSERVE\_CASE(UINT16);

OBSERVE\_CASE(UINT32);

OBSERVE\_CASE(UINT64);

OBSERVE\_CASE(FLOAT16);

OBSERVE\_CASE(FLOAT32);

OBSERVE\_CASE(FLOAT64);

case NPY\_DATETIME:

action = Observe\_DATETIME(descr);

break;

default:

return Status::NotImplemented("Unsupported numpy type ", GetNumPyTypeName(dtype));

}

if (action == INVALID) {

return InvalidMix(dtype);

}

return Status::OK();

}

bool dtype\_was\_observed() const { return current\_type\_num\_ != -1; }

PyArray\_Descr\* current\_dtype() const { return current\_dtype\_; }

int current\_type\_num() const { return current\_type\_num\_; }

private:

int current\_type\_num\_;

PyArray\_Descr\* current\_dtype\_;

};

class TypeInferrer {

// A type inference visitor for Python values

public:

// \param validate\_interval the number of elements to observe before checking

// whether the data is mixed type or has other problems. This helps avoid

// excess computation for each element while also making sure we "bail out"

// early with long sequences that may have problems up front

// \param make\_unions permit mixed-type data by creating union types (not yet

// implemented)

explicit TypeInferrer(bool pandas\_null\_sentinels = false,

int64\_t validate\_interval = 100, bool make\_unions = false)

: pandas\_null\_sentinels\_(pandas\_null\_sentinels),

validate\_interval\_(validate\_interval),

make\_unions\_(make\_unions),

total\_count\_(0),

none\_count\_(0),

bool\_count\_(0),

int\_count\_(0),

date\_count\_(0),

time\_count\_(0),

timestamp\_micro\_count\_(0),

duration\_count\_(0),

float\_count\_(0),

binary\_count\_(0),

unicode\_count\_(0),

decimal\_count\_(0),

list\_count\_(0),

struct\_count\_(0),

arrow\_scalar\_count\_(0),

numpy\_dtype\_count\_(0),

interval\_count\_(0),

max\_decimal\_metadata\_(std::numeric\_limits<int32\_t>::min(),

std::numeric\_limits<int32\_t>::min()),

decimal\_type\_() {

ARROW\_CHECK\_OK(internal::ImportDecimalType(&decimal\_type\_));

ARROW\_CHECK\_OK(ImportPresentIntervalTypes(&interval\_types\_));

}

/// \param[in] obj a Python object in the sequence

/// \param[out] keep\_going if sufficient information has been gathered to

/// attempt to begin converting the sequence, \*keep\_going will be set to true

/// to signal to the calling visitor loop to terminate

Status Visit(PyObject\* obj, bool\* keep\_going) {

++total\_count\_;

if (obj == Py\_None || (pandas\_null\_sentinels\_ && internal::PandasObjectIsNull(obj))) {

++none\_count\_;

} else if (PyBool\_Check(obj)) {

++bool\_count\_;

\*keep\_going = make\_unions\_;

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

++float\_count\_;

\*keep\_going = make\_unions\_;

} else if (internal::IsPyInteger(obj)) {

++int\_count\_;

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

// infer timezone from the first encountered datetime object

if (!timestamp\_micro\_count\_) {

OwnedRef tzinfo(PyObject\_GetAttrString(obj, "tzinfo"));

if (tzinfo.obj() != nullptr && tzinfo.obj() != Py\_None) {

ARROW\_ASSIGN\_OR\_RAISE(timezone\_, internal::TzinfoToString(tzinfo.obj()));

}

}

++timestamp\_micro\_count\_;

\*keep\_going = make\_unions\_;

} else if (PyDelta\_Check(obj)) {

++duration\_count\_;

\*keep\_going = make\_unions\_;

} else if (PyDate\_Check(obj)) {

++date\_count\_;

\*keep\_going = make\_unions\_;

} else if (PyTime\_Check(obj)) {

++time\_count\_;

\*keep\_going = make\_unions\_;

} else if (internal::IsPyBinary(obj)) {

++binary\_count\_;

\*keep\_going = make\_unions\_;

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

++unicode\_count\_;

\*keep\_going = make\_unions\_;

} else if (arrow::py::is\_scalar(obj)) {

RETURN\_NOT\_OK(VisitArrowScalar(obj, keep\_going));

} else if (has\_numpy() && PyArray\_CheckAnyScalarExact(obj)) {

RETURN\_NOT\_OK(VisitDType(PyArray\_DescrFromScalar(obj), keep\_going));

} else if (PySet\_Check(obj) || (Py\_TYPE(obj) == &PyDictValues\_Type)) {

RETURN\_NOT\_OK(VisitSet(obj, keep\_going));

} else if (has\_numpy() && PyArray\_Check(obj)) {

RETURN\_NOT\_OK(VisitNdarray(obj, keep\_going));

} else if (PyDict\_Check(obj)) {

RETURN\_NOT\_OK(VisitDict(obj));

} else if (PyList\_Check(obj) ||

(PyTuple\_Check(obj) &&

!PyObject\_IsInstance(obj, PyTuple\_GetItem(interval\_types\_.obj(), 0)))) {

RETURN\_NOT\_OK(VisitList(obj, keep\_going));

} else if (PyObject\_IsInstance(obj, decimal\_type\_.obj())) {

RETURN\_NOT\_OK(max\_decimal\_metadata\_.Update(obj));

++decimal\_count\_;

} else if (PyObject\_IsInstance(obj, interval\_types\_.obj())) {

++interval\_count\_;

} else {

return internal::InvalidValue(obj,

"did not recognize Python value type when inferring "

"an Arrow data type");

}

if (total\_count\_ % validate\_interval\_ == 0) {

RETURN\_NOT\_OK(Validate());

}

return Status::OK();

}

// Infer value type from a sequence of values

Status VisitSequence(PyObject\* obj, PyObject\* mask = nullptr) {

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

return internal::VisitSequence(

obj, /\*offset=\*/0,

[this](PyObject\* value, bool\* keep\_going) { return Visit(value, keep\_going); });

} else {

return internal::VisitSequenceMasked(

obj, mask, /\*offset=\*/0,

[this](PyObject\* value, uint8\_t masked, bool\* keep\_going) {

if (!masked) {

return Visit(value, keep\_going);

} else {

return Status::OK();

}

});

}

}

// Infer value type from a sequence of values

Status VisitIterable(PyObject\* obj) {

return internal::VisitIterable(obj, [this](PyObject\* value, bool\* keep\_going) {

return Visit(value, keep\_going);

});

}

Status GetType(std::shared\_ptr<DataType>\* out) {

// TODO(wesm): handling forming unions

if (make\_unions\_) {

return Status::NotImplemented("Creating union types not yet supported");

}

RETURN\_NOT\_OK(Validate());

if (arrow\_scalar\_count\_ > 0 && arrow\_scalar\_count\_ + none\_count\_ != total\_count\_) {

return Status::Invalid(

"pyarrow scalars cannot be mixed "

"with other Python scalar values currently");

}

if (numpy\_dtype\_count\_ > 0) {

// All NumPy scalars and Nones/nulls

if (numpy\_dtype\_count\_ + none\_count\_ == total\_count\_) {

return NumPyDtypeToArrow(numpy\_unifier\_.current\_dtype()).Value(out);

}

// The "bad path": data contains a mix of NumPy scalars and

// other kinds of scalars. Note this can happen innocuously

// because numpy.nan is not a NumPy scalar (it's a built-in

// PyFloat)

// TODO(ARROW-5564): Merge together type unification so this

// hack is not necessary

switch (numpy\_unifier\_.current\_type\_num()) {

case NPY\_BOOL:

bool\_count\_ += numpy\_dtype\_count\_;

break;

case NPY\_INT8:

case NPY\_INT16:

case NPY\_INT32:

case NPY\_INT64:

case NPY\_UINT8:

case NPY\_UINT16:

case NPY\_UINT32:

case NPY\_UINT64:

int\_count\_ += numpy\_dtype\_count\_;

break;

case NPY\_FLOAT32:

case NPY\_FLOAT64:

float\_count\_ += numpy\_dtype\_count\_;

break;

case NPY\_DATETIME:

return Status::Invalid(

"numpy.datetime64 scalars cannot be mixed "

"with other Python scalar values currently");

}

}

if (list\_count\_) {

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

RETURN\_NOT\_OK(list\_inferrer\_->GetType(&value\_type));

\*out = list(value\_type);

} else if (struct\_count\_) {

RETURN\_NOT\_OK(GetStructType(out));

} else if (decimal\_count\_) {

if (max\_decimal\_metadata\_.precision() > Decimal128Type::kMaxPrecision) {

// the default constructor does not validate the precision and scale

ARROW\_ASSIGN\_OR\_RAISE(\*out,

Decimal256Type::Make(max\_decimal\_metadata\_.precision(),

max\_decimal\_metadata\_.scale()));

} else {

ARROW\_ASSIGN\_OR\_RAISE(\*out,

Decimal128Type::Make(max\_decimal\_metadata\_.precision(),

max\_decimal\_metadata\_.scale()));

}

} else if (float\_count\_) {

// Prioritize floats before integers

\*out = float64();

} else if (int\_count\_) {

\*out = int64();

} else if (date\_count\_) {

\*out = date32();

} else if (time\_count\_) {

\*out = time64(TimeUnit::MICRO);

} else if (timestamp\_micro\_count\_) {

\*out = timestamp(TimeUnit::MICRO, timezone\_);

} else if (duration\_count\_) {

\*out = duration(TimeUnit::MICRO);

} else if (bool\_count\_) {

\*out = boolean();

} else if (binary\_count\_) {

\*out = binary();

} else if (unicode\_count\_) {

\*out = utf8();

} else if (interval\_count\_) {

\*out = month\_day\_nano\_interval();

} else if (arrow\_scalar\_count\_) {

\*out = scalar\_type\_;

} else {

\*out = null();

}

return Status::OK();

}

int64\_t total\_count() const { return total\_count\_; }

protected:

Status Validate() const {

if (list\_count\_ > 0) {

if (list\_count\_ + none\_count\_ != total\_count\_) {

return Status::Invalid("cannot mix list and non-list, non-null values");

}

RETURN\_NOT\_OK(list\_inferrer\_->Validate());

} else if (struct\_count\_ > 0) {

if (struct\_count\_ + none\_count\_ != total\_count\_) {

return Status::Invalid("cannot mix struct and non-struct, non-null values");

}

for (const auto& it : struct\_inferrers\_) {

RETURN\_NOT\_OK(it.second.Validate());

}

}

return Status::OK();

}

Status VisitArrowScalar(PyObject\* obj, bool\* keep\_going /\* unused \*/) {

ARROW\_ASSIGN\_OR\_RAISE(auto scalar, arrow::py::unwrap\_scalar(obj));

// Check that all the scalar types for the sequence are the same

if (arrow\_scalar\_count\_ > 0 && \*scalar->type != \*scalar\_type\_) {

return internal::InvalidValue(obj, "cannot mix scalars with different types");

}

scalar\_type\_ = scalar->type;

++arrow\_scalar\_count\_;

return Status::OK();

}

Status VisitDType(PyArray\_Descr\* dtype, bool\* keep\_going) {

// Continue visiting dtypes for now.

// TODO(wesm): devise approach for unions

++numpy\_dtype\_count\_;

\*keep\_going = true;

return numpy\_unifier\_.Observe(dtype);

}

Status VisitList(PyObject\* obj, bool\* keep\_going /\* unused \*/) {

if (!list\_inferrer\_) {

list\_inferrer\_.reset(

new TypeInferrer(pandas\_null\_sentinels\_, validate\_interval\_, make\_unions\_));

}

++list\_count\_;

return list\_inferrer\_->VisitSequence(obj);

}

Status VisitSet(PyObject\* obj, bool\* keep\_going /\* unused \*/) {

if (!list\_inferrer\_) {

list\_inferrer\_.reset(

new TypeInferrer(pandas\_null\_sentinels\_, validate\_interval\_, make\_unions\_));

}

++list\_count\_;

return list\_inferrer\_->VisitIterable(obj);

}

Status VisitNdarray(PyObject\* obj, bool\* keep\_going) {

PyArray\_Descr\* dtype = PyArray\_DESCR(reinterpret\_cast<PyArrayObject\*>(obj));

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

return VisitList(obj, keep\_going);

}

// Not an object array: infer child Arrow type from dtype

if (!list\_inferrer\_) {

list\_inferrer\_.reset(

new TypeInferrer(pandas\_null\_sentinels\_, validate\_interval\_, make\_unions\_));

}

++list\_count\_;

// XXX(wesm): In ARROW-4324 I added accounting to check whether

// all of the non-null values have NumPy dtypes, but the

// total\_count not being properly incremented here

++(\*list\_inferrer\_).total\_count\_;

return list\_inferrer\_->VisitDType(dtype, keep\_going);

}

Status VisitDict(PyObject\* obj) {

PyObject\* key\_obj;

PyObject\* value\_obj;

Py\_ssize\_t pos = 0;

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

std::string key;

if (PyUnicode\_Check(key\_obj)) {

RETURN\_NOT\_OK(internal::PyUnicode\_AsStdString(key\_obj, &key));

} else if (PyBytes\_Check(key\_obj)) {

key = internal::PyBytes\_AsStdString(key\_obj);

} else {

return Status::TypeError("Expected dict key of type str or bytes, got '",

Py\_TYPE(key\_obj)->tp\_name, "'");

}

// Get or create visitor for this key

auto it = struct\_inferrers\_.find(key);

if (it == struct\_inferrers\_.end()) {

it = struct\_inferrers\_

.insert(

std::make\_pair(key, TypeInferrer(pandas\_null\_sentinels\_,

validate\_interval\_, make\_unions\_)))

.first;

}

TypeInferrer\* visitor = &it->second;

// We ignore termination signals from child visitors for now

//

// TODO(wesm): keep track of whether type inference has terminated for

// the child visitors to avoid doing unneeded work

bool keep\_going = true;

RETURN\_NOT\_OK(visitor->Visit(value\_obj, &keep\_going));

}

// We do not terminate visiting dicts since we want the union of all

// observed keys

++struct\_count\_;

return Status::OK();

}

Status GetStructType(std::shared\_ptr<DataType>\* out) {

std::vector<std::shared\_ptr<Field>> fields;

for (auto&& it : struct\_inferrers\_) {

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

RETURN\_NOT\_OK(it.second.GetType(&field\_type));

fields.emplace\_back(field(it.first, field\_type));

}

\*out = struct\_(fields);

return Status::OK();

}

private:

bool pandas\_null\_sentinels\_;

int64\_t validate\_interval\_;

bool make\_unions\_;

int64\_t total\_count\_;

int64\_t none\_count\_;

int64\_t bool\_count\_;

int64\_t int\_count\_;

int64\_t date\_count\_;

int64\_t time\_count\_;

int64\_t timestamp\_micro\_count\_;

std::string timezone\_;

int64\_t duration\_count\_;

int64\_t float\_count\_;

int64\_t binary\_count\_;

int64\_t unicode\_count\_;

int64\_t decimal\_count\_;

int64\_t list\_count\_;

int64\_t struct\_count\_;

int64\_t arrow\_scalar\_count\_;

int64\_t numpy\_dtype\_count\_;

int64\_t interval\_count\_;

std::unique\_ptr<TypeInferrer> list\_inferrer\_;

std::map<std::string, TypeInferrer> struct\_inferrers\_;

std::shared\_ptr<DataType> scalar\_type\_;

// If we observe a strongly-typed value in e.g. a NumPy array, we can store

// it here to skip the type counting logic above

NumPyDtypeUnifier numpy\_unifier\_;

internal::DecimalMetadata max\_decimal\_metadata\_;

OwnedRefNoGIL decimal\_type\_;

OwnedRefNoGIL interval\_types\_;

};

// Non-exhaustive type inference

Result<std::shared\_ptr<DataType>> InferArrowType(PyObject\* obj, PyObject\* mask,

bool pandas\_null\_sentinels) {

if (pandas\_null\_sentinels) {

// ARROW-842: If pandas is not installed then null checks will be less

// comprehensive, but that is okay.

internal::InitPandasStaticData();

}

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

TypeInferrer inferrer(pandas\_null\_sentinels);

RETURN\_NOT\_OK(inferrer.VisitSequence(obj, mask));

RETURN\_NOT\_OK(inferrer.GetType(&out\_type));

if (out\_type == nullptr) {

return Status::TypeError("Unable to determine data type");

} else {

return std::move(out\_type);

}

}

ARROW\_PYTHON\_EXPORT

bool IsPyBool(PyObject\* obj) { return internal::PyBoolScalar\_Check(obj); }

ARROW\_PYTHON\_EXPORT

bool IsPyInt(PyObject\* obj) { return internal::PyIntScalar\_Check(obj); }

ARROW\_PYTHON\_EXPORT

bool IsPyFloat(PyObject\* obj) { return internal::PyFloatScalar\_Check(obj); }

} // namespace py

} // namespace arrow