// 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 <algorithm>

#include <limits>

#include "arrow/python/common.h"

#include "arrow/python/decimal.h"

#include "arrow/python/helpers.h"

#include "arrow/type\_fwd.h"

#include "arrow/util/decimal.h"

#include "arrow/util/logging.h"

namespace arrow {

namespace py {

namespace internal {

Status ImportDecimalType(OwnedRef\* decimal\_type) {

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(ImportFromModule(decimal\_module.obj(), "Decimal", decimal\_type));

return Status::OK();

}

Status PythonDecimalToString(PyObject\* python\_decimal, std::string\* out) {

// Call Python's str(decimal\_object)

return PyObject\_StdStringStr(python\_decimal, out);

}

// \brief Infer the precision and scale of a Python decimal.Decimal instance

// \param python\_decimal[in] An instance of decimal.Decimal

// \param precision[out] The value of the inferred precision

// \param scale[out] The value of the inferred scale

// \return The status of the operation

static Status InferDecimalPrecisionAndScale(PyObject\* python\_decimal, int32\_t\* precision,

int32\_t\* scale) {

DCHECK\_NE(python\_decimal, NULLPTR);

DCHECK\_NE(precision, NULLPTR);

DCHECK\_NE(scale, NULLPTR);

// TODO(phillipc): Make sure we perform PyDecimal\_Check(python\_decimal) as a DCHECK

OwnedRef as\_tuple(PyObject\_CallMethod(python\_decimal, const\_cast<char\*>("as\_tuple"),

const\_cast<char\*>("")));

RETURN\_IF\_PYERROR();

DCHECK(PyTuple\_Check(as\_tuple.obj()));

OwnedRef digits(PyObject\_GetAttrString(as\_tuple.obj(), "digits"));

RETURN\_IF\_PYERROR();

DCHECK(PyTuple\_Check(digits.obj()));

const auto num\_digits = static\_cast<int32\_t>(PyTuple\_Size(digits.obj()));

RETURN\_IF\_PYERROR();

OwnedRef py\_exponent(PyObject\_GetAttrString(as\_tuple.obj(), "exponent"));

RETURN\_IF\_PYERROR();

DCHECK(IsPyInteger(py\_exponent.obj()));

const auto exponent = static\_cast<int32\_t>(PyLong\_AsLong(py\_exponent.obj()));

RETURN\_IF\_PYERROR();

if (exponent < 0) {

// If exponent > num\_digits, we have a number with leading zeros

// such as 0.01234. Ensure we have enough precision for leading zeros

// (which are not included in num\_digits).

\*precision = std::max(num\_digits, -exponent);

\*scale = -exponent;

} else {

// Trailing zeros are not included in num\_digits, need to add to precision.

// Note we don't generate negative scales as they are poorly supported

// in non-Arrow systems.

\*precision = num\_digits + exponent;

\*scale = 0;

}

return Status::OK();

}

PyObject\* DecimalFromString(PyObject\* decimal\_constructor,

const std::string& decimal\_string) {

DCHECK\_NE(decimal\_constructor, nullptr);

auto string\_size = decimal\_string.size();

DCHECK\_GT(string\_size, 0);

auto string\_bytes = decimal\_string.c\_str();

DCHECK\_NE(string\_bytes, nullptr);

return PyObject\_CallFunction(decimal\_constructor, const\_cast<char\*>("s#"), string\_bytes,

static\_cast<Py\_ssize\_t>(string\_size));

}

namespace {

template <typename ArrowDecimal>

Status DecimalFromStdString(const std::string& decimal\_string,

const DecimalType& arrow\_type, ArrowDecimal\* out) {

int32\_t inferred\_precision;

int32\_t inferred\_scale;

RETURN\_NOT\_OK(ArrowDecimal::FromString(decimal\_string, out, &inferred\_precision,

&inferred\_scale));

const int32\_t precision = arrow\_type.precision();

const int32\_t scale = arrow\_type.scale();

if (scale != inferred\_scale) {

DCHECK\_NE(out, NULLPTR);

ARROW\_ASSIGN\_OR\_RAISE(\*out, out->Rescale(inferred\_scale, scale));

}

auto inferred\_scale\_delta = inferred\_scale - scale;

if (ARROW\_PREDICT\_FALSE((inferred\_precision - inferred\_scale\_delta) > precision)) {

return Status::Invalid(

"Decimal type with precision ", inferred\_precision,

" does not fit into precision inferred from first array element: ", precision);

}

return Status::OK();

}

template <typename ArrowDecimal>

Status InternalDecimalFromPythonDecimal(PyObject\* python\_decimal,

const DecimalType& arrow\_type,

ArrowDecimal\* out) {

DCHECK\_NE(python\_decimal, NULLPTR);

DCHECK\_NE(out, NULLPTR);

std::string string;

RETURN\_NOT\_OK(PythonDecimalToString(python\_decimal, &string));

return DecimalFromStdString(string, arrow\_type, out);

}

template <typename ArrowDecimal>

Status InternalDecimalFromPyObject(PyObject\* obj, const DecimalType& arrow\_type,

ArrowDecimal\* out) {

DCHECK\_NE(obj, NULLPTR);

DCHECK\_NE(out, NULLPTR);

if (IsPyInteger(obj)) {

// TODO: add a fast path for small-ish ints

std::string string;

RETURN\_NOT\_OK(PyObject\_StdStringStr(obj, &string));

return DecimalFromStdString(string, arrow\_type, out);

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

return InternalDecimalFromPythonDecimal<ArrowDecimal>(obj, arrow\_type, out);

} else {

return Status::TypeError("int or Decimal object expected, got ",

Py\_TYPE(obj)->tp\_name);

}

}

} // namespace

Status DecimalFromPythonDecimal(PyObject\* python\_decimal, const DecimalType& arrow\_type,

Decimal128\* out) {

return InternalDecimalFromPythonDecimal(python\_decimal, arrow\_type, out);

}

Status DecimalFromPyObject(PyObject\* obj, const DecimalType& arrow\_type,

Decimal128\* out) {

return InternalDecimalFromPyObject(obj, arrow\_type, out);

}

Status DecimalFromPythonDecimal(PyObject\* python\_decimal, const DecimalType& arrow\_type,

Decimal256\* out) {

return InternalDecimalFromPythonDecimal(python\_decimal, arrow\_type, out);

}

Status DecimalFromPyObject(PyObject\* obj, const DecimalType& arrow\_type,

Decimal256\* out) {

return InternalDecimalFromPyObject(obj, arrow\_type, out);

}

bool PyDecimal\_Check(PyObject\* obj) {

static OwnedRef decimal\_type;

if (!decimal\_type.obj()) {

ARROW\_CHECK\_OK(ImportDecimalType(&decimal\_type));

DCHECK(PyType\_Check(decimal\_type.obj()));

}

// PyObject\_IsInstance() is slower as it has to check for virtual subclasses

const int result =

PyType\_IsSubtype(Py\_TYPE(obj), reinterpret\_cast<PyTypeObject\*>(decimal\_type.obj()));

ARROW\_CHECK\_NE(result, -1) << " error during PyType\_IsSubtype check";

return result == 1;

}

bool PyDecimal\_ISNAN(PyObject\* obj) {

DCHECK(PyDecimal\_Check(obj)) << "obj is not an instance of decimal.Decimal";

OwnedRef is\_nan(

PyObject\_CallMethod(obj, const\_cast<char\*>("is\_nan"), const\_cast<char\*>("")));

return PyObject\_IsTrue(is\_nan.obj()) == 1;

}

DecimalMetadata::DecimalMetadata()

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

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

DecimalMetadata::DecimalMetadata(int32\_t precision, int32\_t scale)

: precision\_(precision), scale\_(scale) {}

Status DecimalMetadata::Update(int32\_t suggested\_precision, int32\_t suggested\_scale) {

const int32\_t current\_scale = scale\_;

scale\_ = std::max(current\_scale, suggested\_scale);

const int32\_t current\_precision = precision\_;

if (current\_precision == std::numeric\_limits<int32\_t>::min()) {

precision\_ = suggested\_precision;

} else {

auto num\_digits = std::max(current\_precision - current\_scale,

suggested\_precision - suggested\_scale);

precision\_ = std::max(num\_digits + scale\_, current\_precision);

}

return Status::OK();

}

Status DecimalMetadata::Update(PyObject\* object) {

bool is\_decimal = PyDecimal\_Check(object);

if (ARROW\_PREDICT\_FALSE(!is\_decimal || PyDecimal\_ISNAN(object))) {

return Status::OK();

}

int32\_t precision = 0;

int32\_t scale = 0;

RETURN\_NOT\_OK(InferDecimalPrecisionAndScale(object, &precision, &scale));

return Update(precision, scale);

}

} // namespace internal

} // namespace py

} // namespace arrow