// 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/udf.h"

#include "arrow/array/array\_nested.h"

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

#include "arrow/buffer\_builder.h"

#include "arrow/compute/api\_aggregate.h"

#include "arrow/compute/api\_vector.h"

#include "arrow/compute/function.h"

#include "arrow/compute/kernel.h"

#include "arrow/compute/row/grouper.h"

#include "arrow/python/common.h"

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

#include "arrow/table.h"

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

#include "arrow/util/logging.h"

namespace arrow {

using compute::ExecSpan;

using compute::Grouper;

using compute::KernelContext;

using compute::KernelState;

using internal::checked\_cast;

namespace py {

namespace {

struct PythonUdfKernelState : public compute::KernelState {

// NOTE: this KernelState constructor doesn't require the GIL.

// If it did, the corresponding KernelInit::operator() should be wrapped

// within SafeCallIntoPython (GH-43487).

explicit PythonUdfKernelState(std::shared\_ptr<OwnedRefNoGIL> function)

: function(std::move(function)) {}

std::shared\_ptr<OwnedRefNoGIL> function;

};

struct PythonUdfKernelInit {

explicit PythonUdfKernelInit(std::shared\_ptr<OwnedRefNoGIL> function)

: function(std::move(function)) {}

Result<std::unique\_ptr<compute::KernelState>> operator()(

compute::KernelContext\*, const compute::KernelInitArgs&) {

return std::make\_unique<PythonUdfKernelState>(function);

}

std::shared\_ptr<OwnedRefNoGIL> function;

};

struct ScalarUdfAggregator : public compute::KernelState {

virtual Status Consume(compute::KernelContext\* ctx, const compute::ExecSpan& batch) = 0;

virtual Status MergeFrom(compute::KernelContext\* ctx, compute::KernelState&& src) = 0;

virtual Status Finalize(compute::KernelContext\* ctx, Datum\* out) = 0;

};

struct HashUdfAggregator : public compute::KernelState {

virtual Status Resize(KernelContext\* ctx, int64\_t size) = 0;

virtual Status Consume(KernelContext\* ctx, const ExecSpan& batch) = 0;

virtual Status Merge(KernelContext\* ct, KernelState&& other, const ArrayData&) = 0;

virtual Status Finalize(KernelContext\* ctx, Datum\* out) = 0;

};

Status AggregateUdfConsume(compute::KernelContext\* ctx, const compute::ExecSpan& batch) {

return checked\_cast<ScalarUdfAggregator\*>(ctx->state())->Consume(ctx, batch);

}

Status AggregateUdfMerge(compute::KernelContext\* ctx, compute::KernelState&& src,

compute::KernelState\* dst) {

return checked\_cast<ScalarUdfAggregator\*>(dst)->MergeFrom(ctx, std::move(src));

}

Status AggregateUdfFinalize(compute::KernelContext\* ctx, arrow::Datum\* out) {

return checked\_cast<ScalarUdfAggregator\*>(ctx->state())->Finalize(ctx, out);

}

Status HashAggregateUdfResize(KernelContext\* ctx, int64\_t size) {

return checked\_cast<HashUdfAggregator\*>(ctx->state())->Resize(ctx, size);

}

Status HashAggregateUdfConsume(KernelContext\* ctx, const ExecSpan& batch) {

return checked\_cast<HashUdfAggregator\*>(ctx->state())->Consume(ctx, batch);

}

Status HashAggregateUdfMerge(KernelContext\* ctx, KernelState&& src,

const ArrayData& group\_id\_mapping) {

return checked\_cast<HashUdfAggregator\*>(ctx->state())

->Merge(ctx, std::move(src), group\_id\_mapping);

}

Status HashAggregateUdfFinalize(KernelContext\* ctx, Datum\* out) {

return checked\_cast<HashUdfAggregator\*>(ctx->state())->Finalize(ctx, out);

}

struct PythonTableUdfKernelInit {

PythonTableUdfKernelInit(std::shared\_ptr<OwnedRefNoGIL> function\_maker,

UdfWrapperCallback cb)

: function\_maker(std::move(function\_maker)), cb(std::move(cb)) {}

Result<std::unique\_ptr<compute::KernelState>> operator()(

compute::KernelContext\* ctx, const compute::KernelInitArgs&) {

return SafeCallIntoPython(

[this, ctx]() -> Result<std::unique\_ptr<compute::KernelState>> {

UdfContext udf\_context{ctx->memory\_pool(), /\*batch\_length=\*/0};

OwnedRef empty\_tuple(PyTuple\_New(0));

auto function = std::make\_shared<OwnedRefNoGIL>(

cb(function\_maker->obj(), udf\_context, empty\_tuple.obj()));

RETURN\_NOT\_OK(CheckPyError());

if (!PyCallable\_Check(function->obj())) {

return Status::TypeError("Expected a callable Python object.");

}

return std::make\_unique<PythonUdfKernelState>(std::move(function));

});

}

std::shared\_ptr<OwnedRefNoGIL> function\_maker;

UdfWrapperCallback cb;

};

struct PythonUdfScalarAggregatorImpl : public ScalarUdfAggregator {

PythonUdfScalarAggregatorImpl(std::shared\_ptr<OwnedRefNoGIL> function,

UdfWrapperCallback cb,

std::vector<std::shared\_ptr<DataType>> input\_types,

std::shared\_ptr<DataType> output\_type)

: function(std::move(function)),

cb(std::move(cb)),

output\_type(std::move(output\_type)) {

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

for (size\_t i = 0; i < input\_types.size(); i++) {

fields.push\_back(field("", input\_types[i]));

}

input\_schema = schema(std::move(fields));

};

Status Consume(compute::KernelContext\* ctx, const compute::ExecSpan& batch) override {

ARROW\_ASSIGN\_OR\_RAISE(

auto rb, batch.ToExecBatch().ToRecordBatch(input\_schema, ctx->memory\_pool()));

values.push\_back(std::move(rb));

return Status::OK();

}

Status MergeFrom(compute::KernelContext\* ctx, compute::KernelState&& src) override {

auto& other\_values = checked\_cast<PythonUdfScalarAggregatorImpl&>(src).values;

values.insert(values.end(), std::make\_move\_iterator(other\_values.begin()),

std::make\_move\_iterator(other\_values.end()));

other\_values.erase(other\_values.begin(), other\_values.end());

return Status::OK();

}

Status Finalize(compute::KernelContext\* ctx, Datum\* out) override {

auto state =

arrow::internal::checked\_cast<PythonUdfScalarAggregatorImpl\*>(ctx->state());

const int num\_args = input\_schema->num\_fields();

// Note: The way that batches are concatenated together

// would result in using double amount of the memory.

// This is OK for now because non decomposable aggregate

// UDF is supposed to be used with segmented aggregation

// where the size of the segment is more or less constant

// so doubling that is not a big deal. This can be also

// improved in the future to use more efficient way to

// concatenate.

ARROW\_ASSIGN\_OR\_RAISE(auto table,

arrow::Table::FromRecordBatches(input\_schema, values));

ARROW\_ASSIGN\_OR\_RAISE(table, table->CombineChunks(ctx->memory\_pool()));

UdfContext udf\_context{ctx->memory\_pool(), table->num\_rows()};

if (table->num\_rows() == 0) {

return Status::Invalid("Finalized is called with empty inputs");

}

RETURN\_NOT\_OK(SafeCallIntoPython([&] {

std::unique\_ptr<OwnedRef> result;

OwnedRef arg\_tuple(PyTuple\_New(num\_args));

RETURN\_NOT\_OK(CheckPyError());

for (int arg\_id = 0; arg\_id < num\_args; arg\_id++) {

// Since we combined chunks there is only one chunk

std::shared\_ptr<Array> c\_data = table->column(arg\_id)->chunk(0);

PyObject\* data = wrap\_array(c\_data);

PyTuple\_SetItem(arg\_tuple.obj(), arg\_id, data);

}

result =

std::make\_unique<OwnedRef>(cb(function->obj(), udf\_context, arg\_tuple.obj()));

RETURN\_NOT\_OK(CheckPyError());

// unwrapping the output for expected output type

if (is\_scalar(result->obj())) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> val, unwrap\_scalar(result->obj()));

if (\*output\_type != \*val->type) {

return Status::TypeError("Expected output datatype ", output\_type->ToString(),

", but function returned datatype ",

val->type->ToString());

}

out->value = std::move(val);

return Status::OK();

}

return Status::TypeError("Unexpected output type: ",

Py\_TYPE(result->obj())->tp\_name, " (expected Scalar)");

}));

return Status::OK();

}

std::shared\_ptr<OwnedRefNoGIL> function;

UdfWrapperCallback cb;

std::vector<std::shared\_ptr<RecordBatch>> values;

std::shared\_ptr<Schema> input\_schema;

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

};

struct PythonUdfHashAggregatorImpl : public HashUdfAggregator {

PythonUdfHashAggregatorImpl(std::shared\_ptr<OwnedRefNoGIL> function,

UdfWrapperCallback cb,

std::vector<std::shared\_ptr<DataType>> input\_types,

std::shared\_ptr<DataType> output\_type)

: function(std::move(function)),

cb(std::move(cb)),

output\_type(std::move(output\_type)) {

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

fields.reserve(input\_types.size());

for (size\_t i = 0; i < input\_types.size(); i++) {

fields.push\_back(field("", input\_types[i]));

}

input\_schema = schema(std::move(fields));

};

// same as ApplyGrouping in partition.cc

// replicated the code here to avoid complicating the dependencies

static Result<RecordBatchVector> ApplyGroupings(

const ListArray& groupings, const std::shared\_ptr<RecordBatch>& batch) {

ARROW\_ASSIGN\_OR\_RAISE(Datum sorted,

compute::Take(batch, groupings.data()->child\_data[0]));

const auto& sorted\_batch = \*sorted.record\_batch();

RecordBatchVector out(static\_cast<size\_t>(groupings.length()));

for (size\_t i = 0; i < out.size(); ++i) {

out[i] = sorted\_batch.Slice(groupings.value\_offset(i), groupings.value\_length(i));

}

return out;

}

Status Resize(KernelContext\* ctx, int64\_t new\_num\_groups) override {

// We only need to change num\_groups in resize

// similar to other hash aggregate kernels

num\_groups = new\_num\_groups;

return Status::OK();

}

Status Consume(KernelContext\* ctx, const ExecSpan& batch) override {

ARROW\_ASSIGN\_OR\_RAISE(

std::shared\_ptr<RecordBatch> rb,

batch.ToExecBatch().ToRecordBatch(input\_schema, ctx->memory\_pool()));

// This is similar to GroupedListImpl

// last array is the group id

const ArraySpan& groups\_array\_data = batch[batch.num\_values() - 1].array;

DCHECK\_EQ(groups\_array\_data.offset, 0);

int64\_t batch\_num\_values = groups\_array\_data.length;

const auto\* batch\_groups = groups\_array\_data.GetValues<uint32\_t>(1);

RETURN\_NOT\_OK(groups.Append(batch\_groups, batch\_num\_values));

values.push\_back(std::move(rb));

num\_values += batch\_num\_values;

return Status::OK();

}

Status Merge(KernelContext\* ctx, KernelState&& other\_state,

const ArrayData& group\_id\_mapping) override {

// This is similar to GroupedListImpl

auto& other = checked\_cast<PythonUdfHashAggregatorImpl&>(other\_state);

auto& other\_values = other.values;

const uint32\_t\* other\_raw\_groups = other.groups.data();

values.insert(values.end(), std::make\_move\_iterator(other\_values.begin()),

std::make\_move\_iterator(other\_values.end()));

auto g = group\_id\_mapping.GetValues<uint32\_t>(1);

for (uint32\_t other\_g = 0; static\_cast<int64\_t>(other\_g) < other.num\_values;

++other\_g) {

// Different state can have different group\_id mappings, so we

// need to translate the ids

RETURN\_NOT\_OK(groups.Append(g[other\_raw\_groups[other\_g]]));

}

num\_values += other.num\_values;

return Status::OK();

}

Status Finalize(KernelContext\* ctx, Datum\* out) override {

// Exclude the last column which is the group id

const int num\_args = input\_schema->num\_fields() - 1;

ARROW\_ASSIGN\_OR\_RAISE(auto groups\_buffer, groups.Finish());

ARROW\_ASSIGN\_OR\_RAISE(auto groupings,

Grouper::MakeGroupings(UInt32Array(num\_values, groups\_buffer),

static\_cast<uint32\_t>(num\_groups)));

ARROW\_ASSIGN\_OR\_RAISE(auto table,

arrow::Table::FromRecordBatches(input\_schema, values));

ARROW\_ASSIGN\_OR\_RAISE(auto rb, table->CombineChunksToBatch(ctx->memory\_pool()));

UdfContext udf\_context{ctx->memory\_pool(), table->num\_rows()};

if (rb->num\_rows() == 0) {

\*out = Datum();

return Status::OK();

}

ARROW\_ASSIGN\_OR\_RAISE(RecordBatchVector rbs, ApplyGroupings(\*groupings, rb));

return SafeCallIntoPython([&] {

ARROW\_ASSIGN\_OR\_RAISE(std::unique\_ptr<ArrayBuilder> builder,

MakeBuilder(output\_type, ctx->memory\_pool()));

for (auto& group\_rb : rbs) {

std::unique\_ptr<OwnedRef> result;

OwnedRef arg\_tuple(PyTuple\_New(num\_args));

RETURN\_NOT\_OK(CheckPyError());

for (int arg\_id = 0; arg\_id < num\_args; arg\_id++) {

// Since we combined chunks there is only one chunk

std::shared\_ptr<Array> c\_data = group\_rb->column(arg\_id);

PyObject\* data = wrap\_array(c\_data);

PyTuple\_SetItem(arg\_tuple.obj(), arg\_id, data);

}

result =

std::make\_unique<OwnedRef>(cb(function->obj(), udf\_context, arg\_tuple.obj()));

RETURN\_NOT\_OK(CheckPyError());

// unwrapping the output for expected output type

if (is\_scalar(result->obj())) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> val,

unwrap\_scalar(result->obj()));

if (\*output\_type != \*val->type) {

return Status::TypeError("Expected output datatype ", output\_type->ToString(),

", but function returned datatype ",

val->type->ToString());

}

ARROW\_RETURN\_NOT\_OK(builder->AppendScalar(std::move(\*val)));

} else {

return Status::TypeError("Unexpected output type: ",

Py\_TYPE(result->obj())->tp\_name, " (expected Scalar)");

}

}

ARROW\_ASSIGN\_OR\_RAISE(auto result, builder->Finish());

out->value = std::move(result->data());

return Status::OK();

});

}

std::shared\_ptr<OwnedRefNoGIL> function;

UdfWrapperCallback cb;

// Accumulated input batches

std::vector<std::shared\_ptr<RecordBatch>> values;

// Group ids - extracted from the last column from the batch

TypedBufferBuilder<uint32\_t> groups;

int64\_t num\_groups = 0;

int64\_t num\_values = 0;

std::shared\_ptr<Schema> input\_schema;

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

};

struct PythonUdf : public PythonUdfKernelState {

PythonUdf(std::shared\_ptr<OwnedRefNoGIL> function, UdfWrapperCallback cb,

std::vector<TypeHolder> input\_types, compute::OutputType output\_type)

: PythonUdfKernelState(std::move(function)),

cb(std::move(cb)),

input\_types(std::move(input\_types)),

output\_type(std::move(output\_type)) {}

UdfWrapperCallback cb;

std::vector<TypeHolder> input\_types;

compute::OutputType output\_type;

TypeHolder resolved\_type;

Result<TypeHolder> ResolveType(compute::KernelContext\* ctx,

const std::vector<TypeHolder>& types) {

if (input\_types == types) {

if (!resolved\_type) {

ARROW\_ASSIGN\_OR\_RAISE(resolved\_type, output\_type.Resolve(ctx, input\_types));

}

return resolved\_type;

}

return output\_type.Resolve(ctx, types);

}

Status Exec(compute::KernelContext\* ctx, const compute::ExecSpan& batch,

compute::ExecResult\* out) {

auto state = arrow::internal::checked\_cast<PythonUdfKernelState\*>(ctx->state());

PyObject\* function = state->function->obj();

const int num\_args = batch.num\_values();

UdfContext udf\_context{ctx->memory\_pool(), batch.length};

OwnedRef arg\_tuple(PyTuple\_New(num\_args));

RETURN\_NOT\_OK(CheckPyError());

for (int arg\_id = 0; arg\_id < num\_args; arg\_id++) {

if (batch[arg\_id].is\_scalar()) {

std::shared\_ptr<Scalar> c\_data = batch[arg\_id].scalar->GetSharedPtr();

PyObject\* data = wrap\_scalar(c\_data);

PyTuple\_SetItem(arg\_tuple.obj(), arg\_id, data);

} else {

std::shared\_ptr<Array> c\_data = batch[arg\_id].array.ToArray();

PyObject\* data = wrap\_array(c\_data);

PyTuple\_SetItem(arg\_tuple.obj(), arg\_id, data);

}

}

OwnedRef result(cb(function, udf\_context, arg\_tuple.obj()));

RETURN\_NOT\_OK(CheckPyError());

// unwrapping the output for expected output type

if (is\_array(result.obj())) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Array> val, unwrap\_array(result.obj()));

ARROW\_ASSIGN\_OR\_RAISE(TypeHolder type, ResolveType(ctx, batch.GetTypes()));

if (type.type == NULLPTR) {

return Status::TypeError("expected output datatype is null");

}

if (\*type.type != \*val->type()) {

return Status::TypeError("Expected output datatype ", type.type->ToString(),

", but function returned datatype ",

val->type()->ToString());

}

out->value = std::move(val->data());

return Status::OK();

} else {

return Status::TypeError("Unexpected output type: ", Py\_TYPE(result.obj())->tp\_name,

" (expected Array)");

}

return Status::OK();

}

};

Status PythonUdfExec(compute::KernelContext\* ctx, const compute::ExecSpan& batch,

compute::ExecResult\* out) {

auto udf = static\_cast<PythonUdf\*>(ctx->kernel()->data.get());

return SafeCallIntoPython([&]() -> Status { return udf->Exec(ctx, batch, out); });

}

template <class Function, class Kernel>

Status RegisterUdf(PyObject\* function, compute::KernelInit kernel\_init,

UdfWrapperCallback cb, const UdfOptions& options,

compute::FunctionRegistry\* registry) {

if (!PyCallable\_Check(function)) {

return Status::TypeError("Expected a callable Python object.");

}

auto scalar\_func =

std::make\_shared<Function>(options.func\_name, options.arity, options.func\_doc);

std::vector<compute::InputType> input\_types;

for (const auto& in\_dtype : options.input\_types) {

input\_types.emplace\_back(in\_dtype);

}

compute::OutputType output\_type(options.output\_type);

// Take reference before wrapping with OwnedRefNoGIL

Py\_INCREF(function);

auto udf\_data = std::make\_shared<PythonUdf>(

std::make\_shared<OwnedRefNoGIL>(function), cb,

TypeHolder::FromTypes(options.input\_types), options.output\_type);

Kernel kernel(

compute::KernelSignature::Make(std::move(input\_types), std::move(output\_type),

options.arity.is\_varargs),

PythonUdfExec, kernel\_init);

kernel.data = std::move(udf\_data);

kernel.mem\_allocation = compute::MemAllocation::NO\_PREALLOCATE;

kernel.null\_handling = compute::NullHandling::COMPUTED\_NO\_PREALLOCATE;

RETURN\_NOT\_OK(scalar\_func->AddKernel(std::move(kernel)));

if (registry == NULLPTR) {

registry = compute::GetFunctionRegistry();

}

RETURN\_NOT\_OK(registry->AddFunction(std::move(scalar\_func)));

return Status::OK();

}

} // namespace

Status RegisterScalarFunction(PyObject\* function, UdfWrapperCallback cb,

const UdfOptions& options,

compute::FunctionRegistry\* registry) {

return RegisterUdf<compute::ScalarFunction, compute::ScalarKernel>(

function, PythonUdfKernelInit{std::make\_shared<OwnedRefNoGIL>(function)}, cb,

options, registry);

}

Status RegisterVectorFunction(PyObject\* function, UdfWrapperCallback cb,

const UdfOptions& options,

compute::FunctionRegistry\* registry) {

return RegisterUdf<compute::VectorFunction, compute::VectorKernel>(

function, PythonUdfKernelInit{std::make\_shared<OwnedRefNoGIL>(function)}, cb,

options, registry);

}

Status RegisterTabularFunction(PyObject\* function, UdfWrapperCallback cb,

const UdfOptions& options,

compute::FunctionRegistry\* registry) {

if (options.arity.num\_args != 0 || options.arity.is\_varargs) {

return Status::NotImplemented("tabular function of non-null arity");

}

if (options.output\_type->id() != Type::type::STRUCT) {

return Status::Invalid("tabular function with non-struct output");

}

return RegisterUdf<compute::ScalarFunction, compute::ScalarKernel>(

function, PythonTableUdfKernelInit{std::make\_shared<OwnedRefNoGIL>(function), cb},

cb, options, registry);

}

Status RegisterScalarAggregateFunction(PyObject\* function, UdfWrapperCallback cb,

const UdfOptions& options,

compute::FunctionRegistry\* registry) {

if (!PyCallable\_Check(function)) {

return Status::TypeError("Expected a callable Python object.");

}

if (registry == NULLPTR) {

registry = compute::GetFunctionRegistry();

}

static auto default\_scalar\_aggregate\_options =

compute::ScalarAggregateOptions::Defaults();

auto aggregate\_func = std::make\_shared<compute::ScalarAggregateFunction>(

options.func\_name, options.arity, options.func\_doc,

&default\_scalar\_aggregate\_options);

std::vector<compute::InputType> input\_types;

for (const auto& in\_dtype : options.input\_types) {

input\_types.emplace\_back(in\_dtype);

}

compute::OutputType output\_type(options.output\_type);

// Take reference before wrapping with OwnedRefNoGIL

Py\_INCREF(function);

auto function\_ref = std::make\_shared<OwnedRefNoGIL>(function);

compute::KernelInit init = [cb, function\_ref, options](

compute::KernelContext\* ctx,

const compute::KernelInitArgs& args)

-> Result<std::unique\_ptr<compute::KernelState>> {

return std::make\_unique<PythonUdfScalarAggregatorImpl>(

function\_ref, cb, options.input\_types, options.output\_type);

};

auto sig = compute::KernelSignature::Make(

std::move(input\_types), std::move(output\_type), options.arity.is\_varargs);

compute::ScalarAggregateKernel kernel(std::move(sig), std::move(init),

AggregateUdfConsume, AggregateUdfMerge,

AggregateUdfFinalize, /\*ordered=\*/false);

RETURN\_NOT\_OK(aggregate\_func->AddKernel(std::move(kernel)));

RETURN\_NOT\_OK(registry->AddFunction(std::move(aggregate\_func)));

return Status::OK();

}

/// \brief Create a new UdfOptions with adjustment for hash kernel

/// \param options User provided udf options

UdfOptions AdjustForHashAggregate(const UdfOptions& options) {

UdfOptions hash\_options;

// Append hash\_ before the function name to separate from the scalar

// version

hash\_options.func\_name = "hash\_" + options.func\_name;

// Extend input types with group id. Group id is appended by the group

// aggregation node. Here we change both arity and input types

if (options.arity.is\_varargs) {

hash\_options.arity = options.arity;

} else {

hash\_options.arity = compute::Arity(options.arity.num\_args + 1, false);

}

// Changing the function doc shouldn't be necessarily because group id

// is not user visible, however, this is currently needed to pass the

// function validation. The name group\_id\_array is consistent with

// hash kernels in hash\_aggregate.cc

hash\_options.func\_doc = options.func\_doc;

hash\_options.func\_doc.arg\_names.emplace\_back("group\_id\_array");

std::vector<std::shared\_ptr<DataType>> input\_dtypes = options.input\_types;

input\_dtypes.emplace\_back(uint32());

hash\_options.input\_types = std::move(input\_dtypes);

hash\_options.output\_type = options.output\_type;

return hash\_options;

}

Status RegisterHashAggregateFunction(PyObject\* function, UdfWrapperCallback cb,

const UdfOptions& options,

compute::FunctionRegistry\* registry) {

if (!PyCallable\_Check(function)) {

return Status::TypeError("Expected a callable Python object.");

}

if (registry == NULLPTR) {

registry = compute::GetFunctionRegistry();

}

UdfOptions hash\_options = AdjustForHashAggregate(options);

std::vector<compute::InputType> input\_types;

for (const auto& in\_dtype : hash\_options.input\_types) {

input\_types.emplace\_back(in\_dtype);

}

compute::OutputType output\_type(hash\_options.output\_type);

static auto default\_hash\_aggregate\_options =

compute::ScalarAggregateOptions::Defaults();

auto hash\_aggregate\_func = std::make\_shared<compute::HashAggregateFunction>(

hash\_options.func\_name, hash\_options.arity, hash\_options.func\_doc,

&default\_hash\_aggregate\_options);

// Take reference before wrapping with OwnedRefNoGIL

Py\_INCREF(function);

auto function\_ref = std::make\_shared<OwnedRefNoGIL>(function);

compute::KernelInit init = [function\_ref, cb, hash\_options](

compute::KernelContext\* ctx,

const compute::KernelInitArgs& args)

-> Result<std::unique\_ptr<compute::KernelState>> {

return std::make\_unique<PythonUdfHashAggregatorImpl>(

function\_ref, cb, hash\_options.input\_types, hash\_options.output\_type);

};

auto sig = compute::KernelSignature::Make(

std::move(input\_types), std::move(output\_type), hash\_options.arity.is\_varargs);

compute::HashAggregateKernel kernel(

std::move(sig), std::move(init), HashAggregateUdfResize, HashAggregateUdfConsume,

HashAggregateUdfMerge, HashAggregateUdfFinalize, /\*ordered=\*/false);

RETURN\_NOT\_OK(hash\_aggregate\_func->AddKernel(std::move(kernel)));

RETURN\_NOT\_OK(registry->AddFunction(std::move(hash\_aggregate\_func)));

return Status::OK();

}

Status RegisterAggregateFunction(PyObject\* function, UdfWrapperCallback cb,

const UdfOptions& options,

compute::FunctionRegistry\* registry) {

RETURN\_NOT\_OK(RegisterScalarAggregateFunction(function, cb, options, registry));

RETURN\_NOT\_OK(RegisterHashAggregateFunction(function, cb, options, registry));

return Status::OK();

}

Result<std::shared\_ptr<RecordBatchReader>> CallTabularFunction(

const std::string& func\_name, const std::vector<Datum>& args,

compute::FunctionRegistry\* registry) {

if (args.size() != 0) {

return Status::NotImplemented("non-empty arguments to tabular function");

}

if (registry == NULLPTR) {

registry = compute::GetFunctionRegistry();

}

ARROW\_ASSIGN\_OR\_RAISE(auto func, registry->GetFunction(func\_name));

if (func->kind() != compute::Function::SCALAR) {

return Status::Invalid("tabular function of non-scalar kind");

}

auto arity = func->arity();

if (arity.num\_args != 0 || arity.is\_varargs) {

return Status::NotImplemented("tabular function of non-null arity");

}

auto kernels =

arrow::internal::checked\_pointer\_cast<compute::ScalarFunction>(func)->kernels();

if (kernels.size() != 1) {

return Status::NotImplemented("tabular function with non-single kernel");

}

const compute::ScalarKernel\* kernel = kernels[0];

auto out\_type = kernel->signature->out\_type();

if (out\_type.kind() != compute::OutputType::FIXED) {

return Status::Invalid("tabular kernel of non-fixed kind");

}

auto datatype = out\_type.type();

if (datatype->id() != Type::type::STRUCT) {

return Status::Invalid("tabular kernel with non-struct output");

}

auto struct\_type = arrow::internal::checked\_cast<StructType\*>(datatype.get());

auto schema = ::arrow::schema(struct\_type->fields());

std::vector<TypeHolder> in\_types;

ARROW\_ASSIGN\_OR\_RAISE(auto func\_exec,

GetFunctionExecutor(func\_name, in\_types, NULLPTR, registry));

auto next\_func = [schema, func\_exec = std::move(

func\_exec)]() -> Result<std::shared\_ptr<RecordBatch>> {

std::vector<Datum> args;

// passed\_length of -1 or 0 with args.size() of 0 leads to an empty ExecSpanIterator

// in exec.cc and to never invoking the source function, so 1 is passed instead

// TODO: GH-33612: Support batch size in user-defined tabular functions

ARROW\_ASSIGN\_OR\_RAISE(auto datum, func\_exec->Execute(args, /\*passed\_length=\*/1));

if (!datum.is\_array()) {

return Status::Invalid("UDF result of non-array kind");

}

std::shared\_ptr<Array> array = datum.make\_array();

if (array->length() == 0) {

return IterationTraits<std::shared\_ptr<RecordBatch>>::End();

}

ARROW\_ASSIGN\_OR\_RAISE(auto batch, RecordBatch::FromStructArray(std::move(array)));

if (!schema->Equals(batch->schema())) {

return Status::Invalid("UDF result with shape not conforming to schema");

}

return std::move(batch);

};

return RecordBatchReader::MakeFromIterator(MakeFunctionIterator(std::move(next\_func)),

schema);

}

} // namespace py

} // namespace arrow