#include "tensorflow/core/framework/op.h"

#include "tensorflow/core/framework/shape\_inference.h"

#include "tensorflow/core/framework/op\_kernel.h"

#include <twml.h>

#include "tensorflow\_utils.h"

#include "resource\_utils.h"

#include <algorithm>

using std::string;

REGISTER\_OP("GetStringTensorsFromDataRecord")

.Attr("feature\_id: int")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("strings: string")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that decodes and returns string tensors from the data record.

Attr

feature\_id: The hashed id of the feature name.

Input

data\_record\_handle: Resource handle to DataRecord.

Outputs

ids: A 1D int64 tensor representing the input index in a given batch.

strings: A 1D string tensor representing the decoded strings from the batch.

)doc");

REGISTER\_OP("GetStringTensorsFromHashedDataRecord")

.Attr("feature\_id: int")

.Input("hashed\_data\_record\_handle: resource")

.Output("ids: int64")

.Output("strings: string")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that decodes and returns string tensors from the hashed data record.

Attr

feature\_id: The hashed id of the feature name.

Input

data\_record\_handle: Resource handle to DataRecord.

Outputs

ids: A 1D int64 tensor representing the input index in a given batch.

strings: A 1D string tensor representing the decoded strings from the batch.

)doc");

template<typename Resource>

class GetStringTensorsOp : public OpKernel {

private:

int64 feature\_id;

public:

explicit GetStringTensorsOp(OpKernelConstruction \*context)

: OpKernel(context) {

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

}

void Compute(OpKernelContext \*context) override {

auto handle = getHandle<Resource>(context, 0);

const int64 batch\_size = static\_cast<int64>(handle->records.size());

const auto &records = handle->records;

try {

int64 total\_size = 0;

for (const auto &record : records) {

try {

const auto &tensor = record.getRawTensor(feature\_id);

total\_size += static\_cast<int64>(tensor.getNumElements());

} catch(const std::out\_of\_range &err) {

LOG(WARNING) << "Ignoring missing string tensor with key: " << feature\_id << std::endl;

continue;

}

}

twml::ThriftReader reader(nullptr);

TensorShape shape = {total\_size};

Tensor \*strings\_tensor = nullptr;

Tensor \*ids\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, shape, &ids\_tensor));

OP\_REQUIRES\_OK(context, context->allocate\_output(1, shape, &strings\_tensor));

auto strings\_data = strings\_tensor->flat<string>().data();

auto ids\_data = ids\_tensor->flat<int64>().data();

for (int64 i = 0; i < batch\_size; i++) {

const auto &record = records[i];

try {

const twml::RawTensor &tensor = record.getRawTensor(feature\_id);

const uint8\_t \*buffer = static\_cast<const uint8\_t \*>(tensor.getData<void>());

const int64 num\_strings = static\_cast<int64>(tensor.getNumElements());

reader.setBuffer(buffer);

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

const uint8\_t \*curr\_begin = nullptr;

const auto curr\_length = reader.getRawBuffer<uint8\_t>(&curr\_begin);

strings\_data[j] = std::string(curr\_begin, curr\_begin + curr\_length);

ids\_data[j] = i;

}

ids\_data += num\_strings;

strings\_data += num\_strings;

} catch(const std::out\_of\_range &err) {

continue;

}

}

} catch(const std::exception &err) {

context->CtxFailureWithWarning(errors::InvalidArgument(err.what()));

}

}

};

REGISTER\_KERNEL\_BUILDER(

Name("GetStringTensorsFromDataRecord")

.Device(DEVICE\_CPU),

GetStringTensorsOp<DataRecordResource>);

REGISTER\_KERNEL\_BUILDER(

Name("GetStringTensorsFromHashedDataRecord")

.Device(DEVICE\_CPU),

GetStringTensorsOp<HashedDataRecordResource>);

REGISTER\_OP("GetTensorsFromDataRecord")

.Attr("assert\_shape: bool")

.Attr("feature\_id: int")

.Input("data\_record\_handle: resource")

.Output("output: string")

.Output("out\_shape: int64")

.Output("out\_type: string")

.Output("out\_endian: uint8")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that decodes and returns tensors from the data record.

Attr

feature\_id: The hashed id of the feature name.

Input

data\_record\_handle: Resource handle to DataRecord.

Outputs

output: A 2D byte tensor representing the requested feature.

out\_shape: A tensor containing [batch\_size, thrift\_shape].

out\_type: Output type returned as a string tensor of size 1.

out\_endian: Endianness of the bytes returned a tensor of size 1. 0: litte, 1: big.

)doc");

REGISTER\_OP("GetTensorsFromHashedDataRecord")

.Attr("assert\_shape: bool")

.Attr("feature\_id: int")

.Input("hashed\_data\_record\_handle: resource")

.Output("output: string")

.Output("out\_shape: int64")

.Output("out\_type: string")

.Output("out\_endian: uint8")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns decodes and tensors from the hashed data record.

Attr

feature\_id: The hashed id of the feature name.

Input

data\_record\_handle: Resource handle to DataRecord.

Outputs

output: A 2D byte tensor representing the requested feature.

out\_shape: A tensor containing [batch\_size, thrift\_shape].

out\_type: Output type returned as a string tensor of size 1.

out\_endian: Endianness of the bytes returned a tensor of size 1. 0: litte, 1: big.

)doc");

template<class Resource>

class GetTensorsOp : public OpKernel {

private:

bool assert\_shape;

int64 feature\_id;

public:

explicit GetTensorsOp(OpKernelConstruction \*context)

: OpKernel(context), assert\_shape(true) {

OP\_REQUIRES\_OK(context, context->GetAttr("assert\_shape", &assert\_shape));

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

}

void Compute(OpKernelContext \*context) override {

auto handle = getHandle<Resource>(context, 0);

uint64 batch\_size = handle->records.size();

const auto &records = handle->records;

try {

TensorShape raw\_shape = {static\_cast<int64>(batch\_size)};

Tensor\* output\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, raw\_shape, &output\_tensor));

auto output\_flat = output\_tensor->flat<string>();

auto output\_data = output\_flat.data();

twml\_type type = TWML\_TYPE\_UNKNOWN;

bool is\_big\_endian = false;

std::vector<uint64> shape(1, batch\_size);

uint64 length = 0;

for (auto record : records) {

const twml::RawTensor tensor = record.getRawTensor(feature\_id);

const auto &curr\_dims = tensor.getDims();

const auto curr\_type = tensor.getType();

const bool curr\_is\_big\_endian = tensor.is\_big\_endian();

const uint64 curr\_length = tensor.getRawLength();

// Create the output tensor based on first tensor

if (shape.size() == 1) {

// Push the shape of individual tensors into shape

shape.reserve(curr\_dims.size() + 1);

shape.insert(shape.end(), curr\_dims.begin(), curr\_dims.end());

type = curr\_type;

is\_big\_endian = curr\_is\_big\_endian;

length = curr\_length;

} else {

if (assert\_shape) {

// Assert shape of all tensors is the same.

bool is\_same\_shape = std::equal(shape.begin() + 1, shape.end(), curr\_dims.begin());

if (!is\_same\_shape || length != curr\_length) {

throw std::runtime\_error("TensorShape mismatch for feature\_id: "

+ std::to\_string(feature\_id));

}

}

// Assert type and endianness of all tensors is the same.

if (type != curr\_type || is\_big\_endian != curr\_is\_big\_endian) {

throw std::runtime\_error("Tensor type mismatch for feature\_id: "

+ std::to\_string(feature\_id));

}

}

// Copy from datarecord to output

const uint8 \*tensor\_data = reinterpret\_cast<const uint8 \*>(tensor.getData<void>());

\*output\_data = std::string(tensor\_data, tensor\_data + curr\_length);

// Increment it for the next tensor in the batch.

output\_data++;

}

Tensor \*shape\_tensor = nullptr;

TensorShape shape\_shape = {static\_cast<int64>(shape.size())};

OP\_REQUIRES\_OK(context, context->allocate\_output(1, shape\_shape, &shape\_tensor));

auto shape\_flat = shape\_tensor->flat<int64>();

for (int i = 0; i < static\_cast<int>(shape.size()); i++) {

shape\_flat(i) = shape[i];

}

Tensor\* type\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(2, {}, &type\_tensor));

type\_tensor->scalar<string>()() = twml::getTypeName(type);

Tensor\* endian\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(3, {}, &endian\_tensor));

endian\_tensor->scalar<uint8>()() = is\_big\_endian;

} catch(const std::exception &err) {

context->CtxFailureWithWarning(errors::InvalidArgument(err.what()));

}

}

};

REGISTER\_KERNEL\_BUILDER(

Name("GetTensorsFromDataRecord")

.Device(DEVICE\_CPU),

GetTensorsOp<DataRecordResource>);

REGISTER\_KERNEL\_BUILDER(

Name("GetTensorsFromHashedDataRecord")

.Device(DEVICE\_CPU),

GetTensorsOp<HashedDataRecordResource>);

REGISTER\_OP("GetTensorsWithMissingMaskFromDataRecord")

.Attr("assert\_shape: bool")

.Attr("feature\_id: int")

.Attr("default\_shape: list(int)")

.Attr("dtype\_size: int")

.Input("data\_record\_handle: resource")

.Output("output: string")

.Output("out\_type: string")

.Output("out\_endian: uint8")

.Output("is\_found: bool")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that decodes and returns tensors from the data record.

Attr

assert\_shape: Specifies if the shape needs to be same across the batch.

feature\_id: The hashed id of the feature name.

default\_shape: Expected shape of output tensor.

dtype\_size: expected size of each element.

Input

data\_record\_handle: Resource handle to DataRecord.

Outputs

output: A 2D byte tensor representing the requested feature.

out\_type: A string tensor represnting the type.

out\_endian: Endianness of the bytes returned a tensor of size 1. 0: litte, 1: big.

is\_missing: A boolean tensor of length batch\_size represnting if the tensor was found for an input.

)doc");

REGISTER\_OP("GetTensorsWithMissingMaskFromHashedDataRecord")

.Attr("assert\_shape: bool")

.Attr("feature\_id: int")

.Attr("default\_shape: list(int)")

.Attr("dtype\_size: int")

.Input("hashed\_data\_record\_handle: resource")

.Output("output: string")

.Output("out\_type: string")

.Output("out\_endian: uint8")

.Output("is\_found: bool")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that decodes and returns tensors from the data record.

Attr

assert\_shape: Specifies if the shape needs to be same across the batch.

feature\_id: The hashed id of the feature name.

default\_shape: Expected shape of output tensor.

dtype\_size: expected size of each element.

Input

hashed\_data\_record\_handle: Resource handle to HashedDataRecord.

Outputs

output: A 2D byte tensor representing the requested feature.

out\_type: A string tensor represnting the type.

out\_endian: Endianness of the bytes returned a tensor of size 1. 0: litte, 1: big.

is\_missing: A boolean tensor of length batch\_size represnting if the tensor was found for an input.

)doc");

template<class Resource>

class GetTensorsWithMissingMaskOp : public OpKernel {

private:

bool assert\_shape;

int64 feature\_id;

int64 dtype\_size;

std::vector<int64> shape;

public:

explicit GetTensorsWithMissingMaskOp(OpKernelConstruction \*context)

: OpKernel(context), assert\_shape(true) {

OP\_REQUIRES\_OK(context, context->GetAttr("assert\_shape", &assert\_shape));

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_shape", &shape));

OP\_REQUIRES\_OK(context, context->GetAttr("dtype\_size", &dtype\_size));

}

void Compute(OpKernelContext \*context) override {

auto handle = getHandle<Resource>(context, 0);

uint64 batch\_size = handle->records.size();

const auto &records = handle->records;

try {

TensorShape raw\_shape = {static\_cast<int64>(batch\_size)};

Tensor\* output\_tensor = nullptr;

Tensor\* is\_found\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, raw\_shape, &output\_tensor));

OP\_REQUIRES\_OK(context, context->allocate\_output(3, raw\_shape, &is\_found\_tensor));

auto output\_flat = output\_tensor->flat<string>();

auto output\_data = output\_flat.data();

auto is\_found\_data = is\_found\_tensor->flat<bool>().data();

twml\_type type = TWML\_TYPE\_UNKNOWN;

bool is\_big\_endian = false;

uint64 length = std::accumulate(shape.begin(), shape.end(), dtype\_size, std::multiplies<int64>());

for (auto record : records) {

try {

const twml::RawTensor tensor = record.getRawTensor(feature\_id);

const auto &curr\_dims = tensor.getDims();

const auto curr\_type = tensor.getType();

const bool curr\_is\_big\_endian = tensor.is\_big\_endian();

const uint64 curr\_length = tensor.getRawLength();

if (type == TWML\_TYPE\_UNKNOWN) {

type = curr\_type;

is\_big\_endian = curr\_is\_big\_endian;

// FloatTensors are stored as a list of doubles.

// If the requested dtype\_size is 4, update the length.

// NOTE: All the missing tensors before this have wrong length, this is fixed at the end.

if (type == TWML\_TYPE\_DOUBLE && is\_big\_endian && dtype\_size == 4) {

length = length \* 2;

}

} else {

// Assert type and endianness of all tensors is the same.

if (type != curr\_type || is\_big\_endian != curr\_is\_big\_endian) {

throw std::runtime\_error("Tensor type mismatch for feature\_id: "

+ std::to\_string(feature\_id));

}

}

// Assert shape of all tensors is the same.

if (assert\_shape && type != TWML\_TYPE\_UNKNOWN) {

// Assert shape of all tensors is the same.

bool is\_same\_shape = std::equal(shape.begin(), shape.end(), curr\_dims.begin());

if (!is\_same\_shape || length != curr\_length) {

throw std::runtime\_error("TensorShape mismatch for feature\_id: "

+ std::to\_string(feature\_id));

}

}

// Copy from datarecord to output

const uint8 \*tensor\_data = reinterpret\_cast<const uint8 \*>(tensor.getData<void>());

\*output\_data = std::string(tensor\_data, tensor\_data + curr\_length);

\*is\_found\_data = true;

} catch(const std::out\_of\_range &err) {

\*output\_data = std::string();

output\_data->resize(length);

\*is\_found\_data = false;

}

// Increment it for the next tensor in the batch.

output\_data++;

is\_found\_data++;

}

// Reset pointers to the beginning

output\_data = output\_flat.data();

is\_found\_data = is\_found\_tensor->flat<bool>().data();

// Resize any missing tensors before type (and hence true length) was known.

if (type == TWML\_TYPE\_DOUBLE) {

for (int64 i = 0; i < static\_cast<int64>(records.size()); i++) {

if (!is\_found\_data[i]) {

output\_data[i].resize(length);

}

}

}

Tensor\* type\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(1, {}, &type\_tensor));

type\_tensor->scalar<string>()() = twml::getTypeName(type);

Tensor\* endian\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(2, {}, &endian\_tensor));

endian\_tensor->scalar<uint8>()() = is\_big\_endian;

} catch(const std::exception &err) {

context->CtxFailureWithWarning(errors::InvalidArgument(err.what()));

}

}

};

REGISTER\_KERNEL\_BUILDER(

Name("GetTensorsWithMissingMaskFromDataRecord")

.Device(DEVICE\_CPU),

GetTensorsWithMissingMaskOp<DataRecordResource>);

REGISTER\_KERNEL\_BUILDER(

Name("GetTensorsWithMissingMaskFromHashedDataRecord")

.Device(DEVICE\_CPU),

GetTensorsWithMissingMaskOp<HashedDataRecordResource>);

REGISTER\_OP("GetSparseTensorsFromDataRecord")

.Attr("feature\_id: int")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("indices: string")

.Output("values: string")

.Output("dense\_shape: int64")

.Output("values\_type: string")

.Output("valueendian: uint8")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that decodes and returns tensors from the data record.

Attr

feature\_id: The hashed id of the feature name.

Input

data\_record\_handle: Resource handle to DataRecord.

Outputs

ids: A 1D tensor representing which input in the batch the value belongs to.

indices: An string tensor containing indices of the sparse tensor as bytes.

values: An string tensor containing values of the sparse tensor as bytes.

dense\_shape: A tensor containing [batch\_size, thrift\_shape].

values\_type: The data type of value tensor returned as a string tensor of size 1.

values\_endian: Endianness of the bytes returned a tensor of size 1. 0: litte, 1: big.

)doc");

REGISTER\_OP("GetSparseTensorsFromHashedDataRecord")

.Attr("feature\_id: int")

.Input("hashed\_data\_record\_handle: resource")

.Output("ids: int64")

.Output("indices: string")

.Output("values: string")

.Output("dense\_shape: int64")

.Output("values\_type: string")

.Output("values\_endian: uint8")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that decodes and returns tensors from the data record.

Attr

feature\_id: The hashed id of the feature name.

Input

data\_record\_handle: Resource handle to DataRecord.

Outputs

ids: A 1D tensor representing which input in the batch the value belongs to.

indices: An string tensor containing indices of the sparse tensor as bytes.

values: An string tensor containing values of the sparse tensor as bytes.

dense\_shape: A tensor containing [batch\_size, thrift\_shape].

values\_type: The data type of value tensor returned as a string tensor of size 1.

values\_endian: Endianness of the bytes returned a tensor of size 1. 0: litte, 1: big.

)doc");

template<typename Resource>

class GetSparseTensorsOp : public OpKernel {

private:

int64 feature\_id;

public:

explicit GetSparseTensorsOp(OpKernelConstruction \*context)

: OpKernel(context) {

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

}

void Compute(OpKernelContext \*context) override {

auto handle = getHandle<Resource>(context, 0);

const int64 batch\_size = static\_cast<int64>(handle->records.size());

const auto &records = handle->records;

try {

twml\_type type = TWML\_TYPE\_UNKNOWN;

bool is\_big\_endian = false;

std::vector<uint64> shape(1, batch\_size);

int64 total\_length = 0;

std::vector<int64> lengths;

lengths.reserve(batch\_size);

int64 total\_indices\_length = 0;

std::vector<int64> indices\_raw\_lengths;

std::vector<const uint8 \*> indices\_data\_ptrs;

indices\_raw\_lengths.reserve(batch\_size);

indices\_data\_ptrs.reserve(batch\_size);

int64 total\_values\_length = 0;

std::vector<int64> values\_raw\_lengths;

std::vector<const uint8 \*> values\_data\_ptrs;

values\_raw\_lengths.reserve(batch\_size);

values\_data\_ptrs.reserve(batch\_size);

for (auto record : records) {

const twml::RawSparseTensor sparse\_tensor = record.getRawSparseTensor(feature\_id);

const twml::RawTensor indices = sparse\_tensor.indices();

const twml::RawTensor values = sparse\_tensor.values();

const auto &dense\_shape = sparse\_tensor.denseShape();

const auto indices\_type = indices.getType();

const auto indices\_is\_big\_endian = indices.is\_big\_endian();

const auto values\_type = values.getType();

const bool values\_is\_big\_endian = values.is\_big\_endian();

const uint64 indices\_length = indices.getDims().back();

const uint64 values\_length = values.getDims().back();

auto indices\_raw\_length = indices.getRawLength();

auto values\_raw\_length = values.getRawLength();

auto indices\_data\_ptr = reinterpret\_cast<const uint8 \*>(indices.getData<void>());

auto values\_data\_ptr = reinterpret\_cast<const uint8 \*>(values.getData<void>());

indices\_raw\_lengths.push\_back(indices\_raw\_length);

values\_raw\_lengths.push\_back(values\_raw\_length);

indices\_data\_ptrs.push\_back(indices\_data\_ptr);

values\_data\_ptrs.push\_back(values\_data\_ptr);

total\_indices\_length += indices\_raw\_length;

total\_values\_length += values\_raw\_length;

if (shape.size() == 1) {

shape.reserve(dense\_shape.size() + 1);

shape.insert(shape.end(), dense\_shape.begin(), dense\_shape.end());

type = values\_type;

is\_big\_endian = values\_is\_big\_endian;

}

// Assert shape of all tensors is the same.

if (!std::equal(shape.begin() + 1, shape.end(), dense\_shape.begin())) {

throw std::runtime\_error("dense\_shape of sparse tensors doesn't match for feature\_id: "

+ std::to\_string(feature\_id));

}

// Assert type of all values tensor is the same.

if (type != values\_type || is\_big\_endian != values\_is\_big\_endian) {

throw std::runtime\_error("The type of values do not match for feature\_id: "

+ std::to\_string(feature\_id));

}

// Assert indices tensor is big endian and of type INT64.

if (indices\_type != TWML\_TYPE\_INT64 || !indices\_is\_big\_endian) {

throw std::runtime\_error("Unexpected type for index tensor for feature\_id: "

+ std::to\_string(feature\_id));

}

if (indices\_length != values\_length) {

throw std::runtime\_error("The length of values and indices does not match for : "

+ std::to\_string(feature\_id));

}

lengths.push\_back(indices\_length);

total\_length += indices\_length;

}

Tensor\* ids\_tensor = nullptr;

TensorShape ids\_shape = {static\_cast<int64>(total\_length)};

OP\_REQUIRES\_OK(context, context->allocate\_output(0, ids\_shape, &ids\_tensor));

auto ids\_tensor\_flat = ids\_tensor->flat<int64>();

auto ids\_tensor\_data = ids\_tensor\_flat.data();

TensorShape raw\_shape = {static\_cast<int64>(1)};

Tensor\* indices\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(1, raw\_shape, &indices\_tensor));

auto indices\_tensor\_flat = indices\_tensor->flat<string>();

auto indices\_tensor\_string = indices\_tensor\_flat.data();

indices\_tensor\_string->resize(total\_indices\_length);

auto indices\_tensor\_iter = indices\_tensor\_string->begin();

Tensor\* values\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(2, raw\_shape, &values\_tensor));

auto values\_tensor\_flat = values\_tensor->flat<string>();

auto values\_tensor\_string = values\_tensor\_flat.data();

values\_tensor\_string->resize(total\_values\_length);

auto values\_tensor\_iter = values\_tensor\_string->begin();

for (int64 i = 0; i < batch\_size; i++) {

// Fill in the data for id == i for all values in the current input.

std::fill(ids\_tensor\_data, ids\_tensor\_data + lengths[i], i);

ids\_tensor\_data += lengths[i];

indices\_tensor\_iter = std::copy(indices\_data\_ptrs[i],

indices\_data\_ptrs[i] + indices\_raw\_lengths[i],

indices\_tensor\_iter);

values\_tensor\_iter = std::copy(values\_data\_ptrs[i],

values\_data\_ptrs[i] + values\_raw\_lengths[i],

values\_tensor\_iter);

}

Tensor \*shape\_tensor = nullptr;

TensorShape shape\_shape = {static\_cast<int64>(shape.size())};

OP\_REQUIRES\_OK(context, context->allocate\_output(3, shape\_shape, &shape\_tensor));

auto shape\_flat = shape\_tensor->flat<int64>();

for (int i = 0; i < static\_cast<int>(shape.size()); i++) {

shape\_flat(i) = shape[i];

}

Tensor\* type\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(4, {}, &type\_tensor));

type\_tensor->scalar<string>()() = twml::getTypeName(type);

Tensor\* endian\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(5, {}, &endian\_tensor));

endian\_tensor->scalar<uint8>()() = is\_big\_endian;

} catch(const std::exception &err) {

context->CtxFailureWithWarning(errors::InvalidArgument(err.what()));

}

}

};

REGISTER\_KERNEL\_BUILDER(

Name("GetSparseTensorsFromDataRecord")

.Device(DEVICE\_CPU),

GetSparseTensorsOp<DataRecordResource>);

REGISTER\_KERNEL\_BUILDER(

Name("GetSparseTensorsFromHashedDataRecord")

.Device(DEVICE\_CPU),

GetSparseTensorsOp<HashedDataRecordResource>);