// Autogenerated by Thrift Compiler (0.17.0)

// DO NOT EDIT UNLESS YOU ARE SURE THAT YOU KNOW WHAT YOU ARE DOING

#![allow(unused\_imports)]

#![allow(unused\_extern\_crates)]

#![allow(clippy::too\_many\_arguments, clippy::type\_complexity, clippy::vec\_box)]

#![cfg\_attr(rustfmt, rustfmt\_skip)]

use std::cell::RefCell;

use std::collections::{BTreeMap, BTreeSet};

use std::convert::{From, TryFrom};

use std::default::Default;

use std::error::Error;

use std::fmt;

use std::fmt::{Display, Formatter};

use std::rc::Rc;

use thrift::OrderedFloat;

use thrift::{ApplicationError, ApplicationErrorKind, ProtocolError, ProtocolErrorKind, TThriftClient};

use thrift::protocol::{TFieldIdentifier, TListIdentifier, TMapIdentifier, TMessageIdentifier, TMessageType, TInputProtocol, TOutputProtocol, TSerializable, TSetIdentifier, TStructIdentifier, TType};

use thrift::protocol::field\_id;

use thrift::protocol::verify\_expected\_message\_type;

use thrift::protocol::verify\_expected\_sequence\_number;

use thrift::protocol::verify\_expected\_service\_call;

use thrift::protocol::verify\_required\_field\_exists;

use thrift::server::TProcessor;

use crate::tensor;

#[derive(Copy, Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct FeatureType(pub i32);

impl FeatureType {

pub const BINARY: FeatureType = FeatureType(1);

pub const CONTINUOUS: FeatureType = FeatureType(2);

pub const DISCRETE: FeatureType = FeatureType(3);

pub const STRING: FeatureType = FeatureType(4);

pub const SPARSE\_BINARY: FeatureType = FeatureType(5);

pub const SPARSE\_CONTINUOUS: FeatureType = FeatureType(6);

pub const UNKNOWN: FeatureType = FeatureType(7);

pub const BLOB: FeatureType = FeatureType(8);

pub const TENSOR: FeatureType = FeatureType(9);

pub const SPARSE\_TENSOR: FeatureType = FeatureType(10);

pub const FEATURE\_TYPE11: FeatureType = FeatureType(11);

pub const FEATURE\_TYPE12: FeatureType = FeatureType(12);

pub const ENUM\_VALUES: &'static [Self] = &[

Self::BINARY,

Self::CONTINUOUS,

Self::DISCRETE,

Self::STRING,

Self::SPARSE\_BINARY,

Self::SPARSE\_CONTINUOUS,

Self::UNKNOWN,

Self::BLOB,

Self::TENSOR,

Self::SPARSE\_TENSOR,

Self::FEATURE\_TYPE11,

Self::FEATURE\_TYPE12,

];

}

impl TSerializable for FeatureType {

#[allow(clippy::trivially\_copy\_pass\_by\_ref)]

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

o\_prot.write\_i32(self.0)

}

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<FeatureType> {

let enum\_value = i\_prot.read\_i32()?;

Ok(FeatureType::from(enum\_value))

}

}

impl From<i32> for FeatureType {

fn from(i: i32) -> Self {

match i {

1 => FeatureType::BINARY,

2 => FeatureType::CONTINUOUS,

3 => FeatureType::DISCRETE,

4 => FeatureType::STRING,

5 => FeatureType::SPARSE\_BINARY,

6 => FeatureType::SPARSE\_CONTINUOUS,

7 => FeatureType::UNKNOWN,

8 => FeatureType::BLOB,

9 => FeatureType::TENSOR,

10 => FeatureType::SPARSE\_TENSOR,

11 => FeatureType::FEATURE\_TYPE11,

12 => FeatureType::FEATURE\_TYPE12,

\_ => FeatureType(i)

}

}

}

impl From<&i32> for FeatureType {

fn from(i: &i32) -> Self {

FeatureType::from(\*i)

}

}

impl From<FeatureType> for i32 {

fn from(e: FeatureType) -> i32 {

e.0

}

}

impl From<&FeatureType> for i32 {

fn from(e: &FeatureType) -> i32 {

e.0

}

}

//

// DataRecord

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct DataRecord {

pub binary\_features: Option<BTreeSet<i64>>,

pub continuous\_features: Option<BTreeMap<i64, OrderedFloat<f64>>>,

pub discrete\_features: Option<BTreeMap<i64, i64>>,

pub string\_features: Option<BTreeMap<i64, String>>,

pub sparse\_binary\_features: Option<BTreeMap<i64, BTreeSet<String>>>,

pub sparse\_continuous\_features: Option<BTreeMap<i64, BTreeMap<String, OrderedFloat<f64>>>>,

pub blob\_features: Option<BTreeMap<i64, Vec<u8>>>,

pub tensors: Option<BTreeMap<i64, tensor::GeneralTensor>>,

pub sparse\_tensors: Option<BTreeMap<i64, tensor::SparseTensor>>,

}

impl DataRecord {

pub fn new<F1, F2, F3, F4, F5, F6, F7, F8, F9>(binary\_features: F1, continuous\_features: F2, discrete\_features: F3, string\_features: F4, sparse\_binary\_features: F5, sparse\_continuous\_features: F6, blob\_features: F7, tensors: F8, sparse\_tensors: F9) -> DataRecord where F1: Into<Option<BTreeSet<i64>>>, F2: Into<Option<BTreeMap<i64, OrderedFloat<f64>>>>, F3: Into<Option<BTreeMap<i64, i64>>>, F4: Into<Option<BTreeMap<i64, String>>>, F5: Into<Option<BTreeMap<i64, BTreeSet<String>>>>, F6: Into<Option<BTreeMap<i64, BTreeMap<String, OrderedFloat<f64>>>>>, F7: Into<Option<BTreeMap<i64, Vec<u8>>>>, F8: Into<Option<BTreeMap<i64, tensor::GeneralTensor>>>, F9: Into<Option<BTreeMap<i64, tensor::SparseTensor>>> {

DataRecord {

binary\_features: binary\_features.into(),

continuous\_features: continuous\_features.into(),

discrete\_features: discrete\_features.into(),

string\_features: string\_features.into(),

sparse\_binary\_features: sparse\_binary\_features.into(),

sparse\_continuous\_features: sparse\_continuous\_features.into(),

blob\_features: blob\_features.into(),

tensors: tensors.into(),

sparse\_tensors: sparse\_tensors.into(),

}

}

}

impl TSerializable for DataRecord {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<DataRecord> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<BTreeSet<i64>> = None;

let mut f\_2: Option<BTreeMap<i64, OrderedFloat<f64>>> = None;

let mut f\_3: Option<BTreeMap<i64, i64>> = None;

let mut f\_4: Option<BTreeMap<i64, String>> = None;

let mut f\_5: Option<BTreeMap<i64, BTreeSet<String>>> = None;

let mut f\_6: Option<BTreeMap<i64, BTreeMap<String, OrderedFloat<f64>>>> = None;

let mut f\_7: Option<BTreeMap<i64, Vec<u8>>> = None;

let mut f\_8: Option<BTreeMap<i64, tensor::GeneralTensor>> = None;

let mut f\_9: Option<BTreeMap<i64, tensor::SparseTensor>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let set\_ident = i\_prot.read\_set\_begin()?;

let mut val: BTreeSet<i64> = BTreeSet::new();

for \_ in 0..set\_ident.size {

let set\_elem\_0 = i\_prot.read\_i64()?;

val.insert(set\_elem\_0);

}

i\_prot.read\_set\_end()?;

f\_1 = Some(val);

},

2 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, OrderedFloat<f64>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_1 = i\_prot.read\_i64()?;

let map\_val\_2 = OrderedFloat::from(i\_prot.read\_double()?);

val.insert(map\_key\_1, map\_val\_2);

}

i\_prot.read\_map\_end()?;

f\_2 = Some(val);

},

3 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, i64> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_3 = i\_prot.read\_i64()?;

let map\_val\_4 = i\_prot.read\_i64()?;

val.insert(map\_key\_3, map\_val\_4);

}

i\_prot.read\_map\_end()?;

f\_3 = Some(val);

},

4 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, String> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_5 = i\_prot.read\_i64()?;

let map\_val\_6 = i\_prot.read\_string()?;

val.insert(map\_key\_5, map\_val\_6);

}

i\_prot.read\_map\_end()?;

f\_4 = Some(val);

},

5 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeSet<String>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_7 = i\_prot.read\_i64()?;

let set\_ident = i\_prot.read\_set\_begin()?;

let mut map\_val\_8: BTreeSet<String> = BTreeSet::new();

for \_ in 0..set\_ident.size {

let set\_elem\_9 = i\_prot.read\_string()?;

map\_val\_8.insert(set\_elem\_9);

}

i\_prot.read\_set\_end()?;

val.insert(map\_key\_7, map\_val\_8);

}

i\_prot.read\_map\_end()?;

f\_5 = Some(val);

},

6 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeMap<String, OrderedFloat<f64>>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_10 = i\_prot.read\_i64()?;

let map\_ident = i\_prot.read\_map\_begin()?;

let mut map\_val\_11: BTreeMap<String, OrderedFloat<f64>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_12 = i\_prot.read\_string()?;

let map\_val\_13 = OrderedFloat::from(i\_prot.read\_double()?);

map\_val\_11.insert(map\_key\_12, map\_val\_13);

}

i\_prot.read\_map\_end()?;

val.insert(map\_key\_10, map\_val\_11);

}

i\_prot.read\_map\_end()?;

f\_6 = Some(val);

},

7 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, Vec<u8>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_14 = i\_prot.read\_i64()?;

let map\_val\_15 = i\_prot.read\_bytes()?;

val.insert(map\_key\_14, map\_val\_15);

}

i\_prot.read\_map\_end()?;

f\_7 = Some(val);

},

8 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, tensor::GeneralTensor> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_16 = i\_prot.read\_i64()?;

let map\_val\_17 = tensor::GeneralTensor::read\_from\_in\_protocol(i\_prot)?;

val.insert(map\_key\_16, map\_val\_17);

}

i\_prot.read\_map\_end()?;

f\_8 = Some(val);

},

9 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, tensor::SparseTensor> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_18 = i\_prot.read\_i64()?;

let map\_val\_19 = tensor::SparseTensor::read\_from\_in\_protocol(i\_prot)?;

val.insert(map\_key\_18, map\_val\_19);

}

i\_prot.read\_map\_end()?;

f\_9 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

let ret = DataRecord {

binary\_features: f\_1,

continuous\_features: f\_2,

discrete\_features: f\_3,

string\_features: f\_4,

sparse\_binary\_features: f\_5,

sparse\_continuous\_features: f\_6,

blob\_features: f\_7,

tensors: f\_8,

sparse\_tensors: f\_9,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("DataRecord");

o\_prot.write\_struct\_begin(&struct\_ident)?;

if let Some(ref fld\_var) = self.binary\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("binaryFeatures", TType::Set, 1))?;

o\_prot.write\_set\_begin(&TSetIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_set\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.continuous\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("continuousFeatures", TType::Map, 2))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Double, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_double((\*v).into())?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.discrete\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("discreteFeatures", TType::Map, 3))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::I64, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_i64(\*v)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.string\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("stringFeatures", TType::Map, 4))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::String, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_string(v)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_binary\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseBinaryFeatures", TType::Map, 5))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Set, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_set\_begin(&TSetIdentifier::new(TType::String, v.len() as i32))?;

for e in v {

o\_prot.write\_string(e)?;

}

o\_prot.write\_set\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_continuous\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseContinuousFeatures", TType::Map, 6))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Map, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::String, TType::Double, v.len() as i32))?;

for (k, v) in v {

o\_prot.write\_string(k)?;

o\_prot.write\_double((\*v).into())?;

}

o\_prot.write\_map\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.blob\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("blobFeatures", TType::Map, 7))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::String, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_bytes(v)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.tensors {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("tensors", TType::Map, 8))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Struct, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

v.write\_to\_out\_protocol(o\_prot)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_tensors {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseTensors", TType::Map, 9))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Struct, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

v.write\_to\_out\_protocol(o\_prot)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

impl Default for DataRecord {

fn default() -> Self {

DataRecord{

binary\_features: Some(BTreeSet::new()),

continuous\_features: Some(BTreeMap::new()),

discrete\_features: Some(BTreeMap::new()),

string\_features: Some(BTreeMap::new()),

sparse\_binary\_features: Some(BTreeMap::new()),

sparse\_continuous\_features: Some(BTreeMap::new()),

blob\_features: Some(BTreeMap::new()),

tensors: Some(BTreeMap::new()),

sparse\_tensors: Some(BTreeMap::new()),

}

}

}

//

// CompactDataRecord

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct CompactDataRecord {

pub binary\_features: Option<BTreeSet<i64>>,

pub continuous\_features: Option<BTreeMap<i64, i32>>,

pub discrete\_features: Option<BTreeMap<i64, i64>>,

pub string\_features: Option<BTreeMap<i64, String>>,

pub sparse\_binary\_features: Option<BTreeMap<i64, BTreeSet<String>>>,

pub sparse\_binary\_features\_with16b\_sparse\_key: Option<BTreeMap<i64, BTreeSet<i16>>>,

pub sparse\_binary\_features\_with32b\_sparse\_key: Option<BTreeMap<i64, BTreeSet<i32>>>,

pub sparse\_binary\_features\_with64b\_sparse\_key: Option<BTreeMap<i64, BTreeSet<i64>>>,

pub sparse\_continuous\_features: Option<BTreeMap<i64, BTreeMap<String, i32>>>,

pub sparse\_continuous\_features\_with16b\_sparse\_key: Option<BTreeMap<i64, BTreeMap<i16, i32>>>,

pub sparse\_continuous\_features\_with32b\_sparse\_key: Option<BTreeMap<i64, BTreeMap<i32, i32>>>,

pub sparse\_continuous\_features\_with64b\_sparse\_key: Option<BTreeMap<i64, BTreeMap<i64, i32>>>,

pub blob\_features: Option<BTreeMap<i64, Vec<u8>>>,

pub tensors: Option<BTreeMap<i64, tensor::GeneralTensor>>,

pub sparse\_tensors: Option<BTreeMap<i64, tensor::SparseTensor>>,

}

impl CompactDataRecord {

pub fn new<F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15>(binary\_features: F1, continuous\_features: F2, discrete\_features: F3, string\_features: F4, sparse\_binary\_features: F5, sparse\_binary\_features\_with16b\_sparse\_key: F6, sparse\_binary\_features\_with32b\_sparse\_key: F7, sparse\_binary\_features\_with64b\_sparse\_key: F8, sparse\_continuous\_features: F9, sparse\_continuous\_features\_with16b\_sparse\_key: F10, sparse\_continuous\_features\_with32b\_sparse\_key: F11, sparse\_continuous\_features\_with64b\_sparse\_key: F12, blob\_features: F13, tensors: F14, sparse\_tensors: F15) -> CompactDataRecord where F1: Into<Option<BTreeSet<i64>>>, F2: Into<Option<BTreeMap<i64, i32>>>, F3: Into<Option<BTreeMap<i64, i64>>>, F4: Into<Option<BTreeMap<i64, String>>>, F5: Into<Option<BTreeMap<i64, BTreeSet<String>>>>, F6: Into<Option<BTreeMap<i64, BTreeSet<i16>>>>, F7: Into<Option<BTreeMap<i64, BTreeSet<i32>>>>, F8: Into<Option<BTreeMap<i64, BTreeSet<i64>>>>, F9: Into<Option<BTreeMap<i64, BTreeMap<String, i32>>>>, F10: Into<Option<BTreeMap<i64, BTreeMap<i16, i32>>>>, F11: Into<Option<BTreeMap<i64, BTreeMap<i32, i32>>>>, F12: Into<Option<BTreeMap<i64, BTreeMap<i64, i32>>>>, F13: Into<Option<BTreeMap<i64, Vec<u8>>>>, F14: Into<Option<BTreeMap<i64, tensor::GeneralTensor>>>, F15: Into<Option<BTreeMap<i64, tensor::SparseTensor>>> {

CompactDataRecord {

binary\_features: binary\_features.into(),

continuous\_features: continuous\_features.into(),

discrete\_features: discrete\_features.into(),

string\_features: string\_features.into(),

sparse\_binary\_features: sparse\_binary\_features.into(),

sparse\_binary\_features\_with16b\_sparse\_key: sparse\_binary\_features\_with16b\_sparse\_key.into(),

sparse\_binary\_features\_with32b\_sparse\_key: sparse\_binary\_features\_with32b\_sparse\_key.into(),

sparse\_binary\_features\_with64b\_sparse\_key: sparse\_binary\_features\_with64b\_sparse\_key.into(),

sparse\_continuous\_features: sparse\_continuous\_features.into(),

sparse\_continuous\_features\_with16b\_sparse\_key: sparse\_continuous\_features\_with16b\_sparse\_key.into(),

sparse\_continuous\_features\_with32b\_sparse\_key: sparse\_continuous\_features\_with32b\_sparse\_key.into(),

sparse\_continuous\_features\_with64b\_sparse\_key: sparse\_continuous\_features\_with64b\_sparse\_key.into(),

blob\_features: blob\_features.into(),

tensors: tensors.into(),

sparse\_tensors: sparse\_tensors.into(),

}

}

}

impl TSerializable for CompactDataRecord {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<CompactDataRecord> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<BTreeSet<i64>> = None;

let mut f\_2: Option<BTreeMap<i64, i32>> = None;

let mut f\_3: Option<BTreeMap<i64, i64>> = None;

let mut f\_4: Option<BTreeMap<i64, String>> = None;

let mut f\_5: Option<BTreeMap<i64, BTreeSet<String>>> = None;

let mut f\_6: Option<BTreeMap<i64, BTreeSet<i16>>> = None;

let mut f\_7: Option<BTreeMap<i64, BTreeSet<i32>>> = None;

let mut f\_8: Option<BTreeMap<i64, BTreeSet<i64>>> = None;

let mut f\_9: Option<BTreeMap<i64, BTreeMap<String, i32>>> = None;

let mut f\_10: Option<BTreeMap<i64, BTreeMap<i16, i32>>> = None;

let mut f\_11: Option<BTreeMap<i64, BTreeMap<i32, i32>>> = None;

let mut f\_12: Option<BTreeMap<i64, BTreeMap<i64, i32>>> = None;

let mut f\_13: Option<BTreeMap<i64, Vec<u8>>> = None;

let mut f\_14: Option<BTreeMap<i64, tensor::GeneralTensor>> = None;

let mut f\_15: Option<BTreeMap<i64, tensor::SparseTensor>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let set\_ident = i\_prot.read\_set\_begin()?;

let mut val: BTreeSet<i64> = BTreeSet::new();

for \_ in 0..set\_ident.size {

let set\_elem\_20 = i\_prot.read\_i64()?;

val.insert(set\_elem\_20);

}

i\_prot.read\_set\_end()?;

f\_1 = Some(val);

},

2 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, i32> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_21 = i\_prot.read\_i64()?;

let map\_val\_22 = i\_prot.read\_i32()?;

val.insert(map\_key\_21, map\_val\_22);

}

i\_prot.read\_map\_end()?;

f\_2 = Some(val);

},

3 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, i64> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_23 = i\_prot.read\_i64()?;

let map\_val\_24 = i\_prot.read\_i64()?;

val.insert(map\_key\_23, map\_val\_24);

}

i\_prot.read\_map\_end()?;

f\_3 = Some(val);

},

4 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, String> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_25 = i\_prot.read\_i64()?;

let map\_val\_26 = i\_prot.read\_string()?;

val.insert(map\_key\_25, map\_val\_26);

}

i\_prot.read\_map\_end()?;

f\_4 = Some(val);

},

5 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeSet<String>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_27 = i\_prot.read\_i64()?;

let set\_ident = i\_prot.read\_set\_begin()?;

let mut map\_val\_28: BTreeSet<String> = BTreeSet::new();

for \_ in 0..set\_ident.size {

let set\_elem\_29 = i\_prot.read\_string()?;

map\_val\_28.insert(set\_elem\_29);

}

i\_prot.read\_set\_end()?;

val.insert(map\_key\_27, map\_val\_28);

}

i\_prot.read\_map\_end()?;

f\_5 = Some(val);

},

6 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeSet<i16>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_30 = i\_prot.read\_i64()?;

let set\_ident = i\_prot.read\_set\_begin()?;

let mut map\_val\_31: BTreeSet<i16> = BTreeSet::new();

for \_ in 0..set\_ident.size {

let set\_elem\_32 = i\_prot.read\_i16()?;

map\_val\_31.insert(set\_elem\_32);

}

i\_prot.read\_set\_end()?;

val.insert(map\_key\_30, map\_val\_31);

}

i\_prot.read\_map\_end()?;

f\_6 = Some(val);

},

7 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeSet<i32>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_33 = i\_prot.read\_i64()?;

let set\_ident = i\_prot.read\_set\_begin()?;

let mut map\_val\_34: BTreeSet<i32> = BTreeSet::new();

for \_ in 0..set\_ident.size {

let set\_elem\_35 = i\_prot.read\_i32()?;

map\_val\_34.insert(set\_elem\_35);

}

i\_prot.read\_set\_end()?;

val.insert(map\_key\_33, map\_val\_34);

}

i\_prot.read\_map\_end()?;

f\_7 = Some(val);

},

8 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeSet<i64>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_36 = i\_prot.read\_i64()?;

let set\_ident = i\_prot.read\_set\_begin()?;

let mut map\_val\_37: BTreeSet<i64> = BTreeSet::new();

for \_ in 0..set\_ident.size {

let set\_elem\_38 = i\_prot.read\_i64()?;

map\_val\_37.insert(set\_elem\_38);

}

i\_prot.read\_set\_end()?;

val.insert(map\_key\_36, map\_val\_37);

}

i\_prot.read\_map\_end()?;

f\_8 = Some(val);

},

9 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeMap<String, i32>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_39 = i\_prot.read\_i64()?;

let map\_ident = i\_prot.read\_map\_begin()?;

let mut map\_val\_40: BTreeMap<String, i32> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_41 = i\_prot.read\_string()?;

let map\_val\_42 = i\_prot.read\_i32()?;

map\_val\_40.insert(map\_key\_41, map\_val\_42);

}

i\_prot.read\_map\_end()?;

val.insert(map\_key\_39, map\_val\_40);

}

i\_prot.read\_map\_end()?;

f\_9 = Some(val);

},

10 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeMap<i16, i32>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_43 = i\_prot.read\_i64()?;

let map\_ident = i\_prot.read\_map\_begin()?;

let mut map\_val\_44: BTreeMap<i16, i32> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_45 = i\_prot.read\_i16()?;

let map\_val\_46 = i\_prot.read\_i32()?;

map\_val\_44.insert(map\_key\_45, map\_val\_46);

}

i\_prot.read\_map\_end()?;

val.insert(map\_key\_43, map\_val\_44);

}

i\_prot.read\_map\_end()?;

f\_10 = Some(val);

},

11 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeMap<i32, i32>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_47 = i\_prot.read\_i64()?;

let map\_ident = i\_prot.read\_map\_begin()?;

let mut map\_val\_48: BTreeMap<i32, i32> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_49 = i\_prot.read\_i32()?;

let map\_val\_50 = i\_prot.read\_i32()?;

map\_val\_48.insert(map\_key\_49, map\_val\_50);

}

i\_prot.read\_map\_end()?;

val.insert(map\_key\_47, map\_val\_48);

}

i\_prot.read\_map\_end()?;

f\_11 = Some(val);

},

12 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, BTreeMap<i64, i32>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_51 = i\_prot.read\_i64()?;

let map\_ident = i\_prot.read\_map\_begin()?;

let mut map\_val\_52: BTreeMap<i64, i32> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_53 = i\_prot.read\_i64()?;

let map\_val\_54 = i\_prot.read\_i32()?;

map\_val\_52.insert(map\_key\_53, map\_val\_54);

}

i\_prot.read\_map\_end()?;

val.insert(map\_key\_51, map\_val\_52);

}

i\_prot.read\_map\_end()?;

f\_12 = Some(val);

},

13 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, Vec<u8>> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_55 = i\_prot.read\_i64()?;

let map\_val\_56 = i\_prot.read\_bytes()?;

val.insert(map\_key\_55, map\_val\_56);

}

i\_prot.read\_map\_end()?;

f\_13 = Some(val);

},

14 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, tensor::GeneralTensor> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_57 = i\_prot.read\_i64()?;

let map\_val\_58 = tensor::GeneralTensor::read\_from\_in\_protocol(i\_prot)?;

val.insert(map\_key\_57, map\_val\_58);

}

i\_prot.read\_map\_end()?;

f\_14 = Some(val);

},

15 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, tensor::SparseTensor> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_59 = i\_prot.read\_i64()?;

let map\_val\_60 = tensor::SparseTensor::read\_from\_in\_protocol(i\_prot)?;

val.insert(map\_key\_59, map\_val\_60);

}

i\_prot.read\_map\_end()?;

f\_15 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

let ret = CompactDataRecord {

binary\_features: f\_1,

continuous\_features: f\_2,

discrete\_features: f\_3,

string\_features: f\_4,

sparse\_binary\_features: f\_5,

sparse\_binary\_features\_with16b\_sparse\_key: f\_6,

sparse\_binary\_features\_with32b\_sparse\_key: f\_7,

sparse\_binary\_features\_with64b\_sparse\_key: f\_8,

sparse\_continuous\_features: f\_9,

sparse\_continuous\_features\_with16b\_sparse\_key: f\_10,

sparse\_continuous\_features\_with32b\_sparse\_key: f\_11,

sparse\_continuous\_features\_with64b\_sparse\_key: f\_12,

blob\_features: f\_13,

tensors: f\_14,

sparse\_tensors: f\_15,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("CompactDataRecord");

o\_prot.write\_struct\_begin(&struct\_ident)?;

if let Some(ref fld\_var) = self.binary\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("binaryFeatures", TType::Set, 1))?;

o\_prot.write\_set\_begin(&TSetIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_set\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.continuous\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("continuousFeatures", TType::Map, 2))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::I32, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_i32(\*v)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.discrete\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("discreteFeatures", TType::Map, 3))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::I64, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_i64(\*v)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.string\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("stringFeatures", TType::Map, 4))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::String, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_string(v)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_binary\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseBinaryFeatures", TType::Map, 5))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Set, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_set\_begin(&TSetIdentifier::new(TType::String, v.len() as i32))?;

for e in v {

o\_prot.write\_string(e)?;

}

o\_prot.write\_set\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_binary\_features\_with16b\_sparse\_key {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseBinaryFeaturesWith16bSparseKey", TType::Map, 6))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Set, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_set\_begin(&TSetIdentifier::new(TType::I16, v.len() as i32))?;

for e in v {

o\_prot.write\_i16(\*e)?;

}

o\_prot.write\_set\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_binary\_features\_with32b\_sparse\_key {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseBinaryFeaturesWith32bSparseKey", TType::Map, 7))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Set, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_set\_begin(&TSetIdentifier::new(TType::I32, v.len() as i32))?;

for e in v {

o\_prot.write\_i32(\*e)?;

}

o\_prot.write\_set\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_binary\_features\_with64b\_sparse\_key {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseBinaryFeaturesWith64bSparseKey", TType::Map, 8))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Set, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_set\_begin(&TSetIdentifier::new(TType::I64, v.len() as i32))?;

for e in v {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_set\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_continuous\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseContinuousFeatures", TType::Map, 9))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Map, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::String, TType::I32, v.len() as i32))?;

for (k, v) in v {

o\_prot.write\_string(k)?;

o\_prot.write\_i32(\*v)?;

}

o\_prot.write\_map\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_continuous\_features\_with16b\_sparse\_key {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseContinuousFeaturesWith16bSparseKey", TType::Map, 10))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Map, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I16, TType::I32, v.len() as i32))?;

for (k, v) in v {

o\_prot.write\_i16(\*k)?;

o\_prot.write\_i32(\*v)?;

}

o\_prot.write\_map\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_continuous\_features\_with32b\_sparse\_key {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseContinuousFeaturesWith32bSparseKey", TType::Map, 11))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Map, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I32, TType::I32, v.len() as i32))?;

for (k, v) in v {

o\_prot.write\_i32(\*k)?;

o\_prot.write\_i32(\*v)?;

}

o\_prot.write\_map\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_continuous\_features\_with64b\_sparse\_key {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseContinuousFeaturesWith64bSparseKey", TType::Map, 12))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Map, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::I32, v.len() as i32))?;

for (k, v) in v {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_i32(\*v)?;

}

o\_prot.write\_map\_end()?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.blob\_features {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("blobFeatures", TType::Map, 13))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::String, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

o\_prot.write\_bytes(v)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.tensors {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("tensors", TType::Map, 14))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Struct, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

v.write\_to\_out\_protocol(o\_prot)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_tensors {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseTensors", TType::Map, 15))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Struct, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

v.write\_to\_out\_protocol(o\_prot)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

impl Default for CompactDataRecord {

fn default() -> Self {

CompactDataRecord{

binary\_features: Some(BTreeSet::new()),

continuous\_features: Some(BTreeMap::new()),

discrete\_features: Some(BTreeMap::new()),

string\_features: Some(BTreeMap::new()),

sparse\_binary\_features: Some(BTreeMap::new()),

sparse\_binary\_features\_with16b\_sparse\_key: Some(BTreeMap::new()),

sparse\_binary\_features\_with32b\_sparse\_key: Some(BTreeMap::new()),

sparse\_binary\_features\_with64b\_sparse\_key: Some(BTreeMap::new()),

sparse\_continuous\_features: Some(BTreeMap::new()),

sparse\_continuous\_features\_with16b\_sparse\_key: Some(BTreeMap::new()),

sparse\_continuous\_features\_with32b\_sparse\_key: Some(BTreeMap::new()),

sparse\_continuous\_features\_with64b\_sparse\_key: Some(BTreeMap::new()),

blob\_features: Some(BTreeMap::new()),

tensors: Some(BTreeMap::new()),

sparse\_tensors: Some(BTreeMap::new()),

}

}

}

//

// TensorRecord

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct TensorRecord {

pub tensors: Option<BTreeMap<i64, tensor::GeneralTensor>>,

pub sparse\_tensors: Option<BTreeMap<i64, tensor::SparseTensor>>,

}

impl TensorRecord {

pub fn new<F1, F2>(tensors: F1, sparse\_tensors: F2) -> TensorRecord where F1: Into<Option<BTreeMap<i64, tensor::GeneralTensor>>>, F2: Into<Option<BTreeMap<i64, tensor::SparseTensor>>> {

TensorRecord {

tensors: tensors.into(),

sparse\_tensors: sparse\_tensors.into(),

}

}

}

impl TSerializable for TensorRecord {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<TensorRecord> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<BTreeMap<i64, tensor::GeneralTensor>> = None;

let mut f\_2: Option<BTreeMap<i64, tensor::SparseTensor>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, tensor::GeneralTensor> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_61 = i\_prot.read\_i64()?;

let map\_val\_62 = tensor::GeneralTensor::read\_from\_in\_protocol(i\_prot)?;

val.insert(map\_key\_61, map\_val\_62);

}

i\_prot.read\_map\_end()?;

f\_1 = Some(val);

},

2 => {

let map\_ident = i\_prot.read\_map\_begin()?;

let mut val: BTreeMap<i64, tensor::SparseTensor> = BTreeMap::new();

for \_ in 0..map\_ident.size {

let map\_key\_63 = i\_prot.read\_i64()?;

let map\_val\_64 = tensor::SparseTensor::read\_from\_in\_protocol(i\_prot)?;

val.insert(map\_key\_63, map\_val\_64);

}

i\_prot.read\_map\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

let ret = TensorRecord {

tensors: f\_1,

sparse\_tensors: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("TensorRecord");

o\_prot.write\_struct\_begin(&struct\_ident)?;

if let Some(ref fld\_var) = self.tensors {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("tensors", TType::Map, 1))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Struct, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

v.write\_to\_out\_protocol(o\_prot)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.sparse\_tensors {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("sparseTensors", TType::Map, 2))?;

o\_prot.write\_map\_begin(&TMapIdentifier::new(TType::I64, TType::Struct, fld\_var.len() as i32))?;

for (k, v) in fld\_var {

o\_prot.write\_i64(\*k)?;

v.write\_to\_out\_protocol(o\_prot)?;

}

o\_prot.write\_map\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

impl Default for TensorRecord {

fn default() -> Self {

TensorRecord{

tensors: Some(BTreeMap::new()),

sparse\_tensors: Some(BTreeMap::new()),

}

}

}

//

// FeatureMetaInfo

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct FeatureMetaInfo {

pub feature\_id: Option<i64>,

pub full\_feature\_name: Option<String>,

pub feature\_type: Option<FeatureType>,

}

impl FeatureMetaInfo {

pub fn new<F1, F2, F3>(feature\_id: F1, full\_feature\_name: F2, feature\_type: F3) -> FeatureMetaInfo where F1: Into<Option<i64>>, F2: Into<Option<String>>, F3: Into<Option<FeatureType>> {

FeatureMetaInfo {

feature\_id: feature\_id.into(),

full\_feature\_name: full\_feature\_name.into(),

feature\_type: feature\_type.into(),

}

}

}

impl TSerializable for FeatureMetaInfo {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<FeatureMetaInfo> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<i64> = None;

let mut f\_2: Option<String> = None;

let mut f\_3: Option<FeatureType> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let val = i\_prot.read\_i64()?;

f\_1 = Some(val);

},

2 => {

let val = i\_prot.read\_string()?;

f\_2 = Some(val);

},

3 => {

let val = FeatureType::read\_from\_in\_protocol(i\_prot)?;

f\_3 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

let ret = FeatureMetaInfo {

feature\_id: f\_1,

full\_feature\_name: f\_2,

feature\_type: f\_3,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("FeatureMetaInfo");

o\_prot.write\_struct\_begin(&struct\_ident)?;

if let Some(fld\_var) = self.feature\_id {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("featureId", TType::I64, 1))?;

o\_prot.write\_i64(fld\_var)?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.full\_feature\_name {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("fullFeatureName", TType::String, 2))?;

o\_prot.write\_string(fld\_var)?;

o\_prot.write\_field\_end()?

}

if let Some(ref fld\_var) = self.feature\_type {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("featureType", TType::I32, 3))?;

fld\_var.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

impl Default for FeatureMetaInfo {

fn default() -> Self {

FeatureMetaInfo{

feature\_id: Some(0),

full\_feature\_name: Some("".to\_owned()),

feature\_type: None,

}

}

}

//

// FeatureMetaInfoList

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct FeatureMetaInfoList {

pub contents: Option<Vec<FeatureMetaInfo>>,

}

impl FeatureMetaInfoList {

pub fn new<F1>(contents: F1) -> FeatureMetaInfoList where F1: Into<Option<Vec<FeatureMetaInfo>>> {

FeatureMetaInfoList {

contents: contents.into(),

}

}

}

impl TSerializable for FeatureMetaInfoList {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<FeatureMetaInfoList> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<FeatureMetaInfo>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<FeatureMetaInfo> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_65 = FeatureMetaInfo::read\_from\_in\_protocol(i\_prot)?;

val.push(list\_elem\_65);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

let ret = FeatureMetaInfoList {

contents: f\_1,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("FeatureMetaInfoList");

o\_prot.write\_struct\_begin(&struct\_ident)?;

if let Some(ref fld\_var) = self.contents {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("contents", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::Struct, fld\_var.len() as i32))?;

for e in fld\_var {

e.write\_to\_out\_protocol(o\_prot)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

impl Default for FeatureMetaInfoList {

fn default() -> Self {

FeatureMetaInfoList{

contents: Some(Vec::new()),

}

}

}