#include "internal/thrift.h"

#include "internal/error.h"

#include <string>

#include <cmath>

#include <twml/DataRecordReader.h>

namespace twml {

inline std::string bufferToString(int32\_t str\_len, const uint8\_t \*str) {

return std::string(str, str + str\_len);

}

bool DataRecordReader::keepKey(const int64\_t &key, int64\_t &code) {

auto it = m\_keep\_map->find(key);

if (it == m\_keep\_map->end()) return false;

code = it->second;

return true;

}

bool DataRecordReader::isLabel(const int64\_t &key, int64\_t &code) {

if (m\_labels\_map == nullptr) return false;

auto it = m\_labels\_map->find(key);

if (it == m\_labels\_map->end()) return false;

code = it->second;

return true;

}

bool DataRecordReader::isWeight(const int64\_t &key, int64\_t &code) {

if (m\_weights\_map == nullptr) return false;

auto it = m\_weights\_map->find(key);

if (it == m\_weights\_map->end()) return false;

code = it->second;

return true;

}

void DataRecordReader::readBinary(

const int feature\_type,

DataRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_SET, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

int32\_t length = readInt32();

int64\_t id, code;

#ifdef USE\_DENSE\_HASH

record->m\_binary.resize(2 \* length);

#else

record->m\_binary.reserve(2 \* length);

#endif

for (int32\_t i = 0; i < length; i++) {

id = readInt64();

record->m\_binary.insert(id);

if (isLabel(id, code)) {

record->addLabel(code);

}

}

}

void DataRecordReader::readContinuous(

const int feature\_type,

DataRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_DOUBLE, "value\_type");

int32\_t length = readInt32();

int64\_t id, code;

#ifdef USE\_DENSE\_HASH

record->m\_continuous.resize(2 \* length);

#else

record->m\_continuous.reserve(2 \* length);

#endif

for (int32\_t i = 0; i < length; i++) {

id = readInt64();

double val = readDouble();

if (!std::isnan(val)) {

record->m\_continuous[id] = val;

}

if (isLabel(id, code)) {

record->addLabel(code, val);

} else if (isWeight(id, code)) {

record->addWeight(code, val);

}

}

}

void DataRecordReader::readDiscrete(

const int feature\_type,

DataRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "value\_type");

int32\_t length = readInt32();

int64\_t id;

#ifdef USE\_DENSE\_HASH

record->m\_discrete.resize(2 \* length);

#else

record->m\_discrete.reserve(2 \* length);

#endif

for (int32\_t i = 0; i < length; i++) {

id = readInt64();

record->m\_discrete[id] = readInt64();

}

}

void DataRecordReader::readString(

const int feature\_type,

DataRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRING, "value\_type");

int32\_t length = readInt32();

int64\_t id;

#ifdef USE\_DENSE\_HASH

record->m\_string.resize(2 \* length);

#else

record->m\_string.reserve(2 \* length);

#endif

for (int32\_t i = 0; i < length; i++) {

id = readInt64();

const uint8\_t \*begin = nullptr;

int32\_t str\_len = getRawBuffer<uint8\_t>(&begin);

record->m\_string[id] = bufferToString(str\_len, begin);

}

}

void DataRecordReader::readSparseBinary(

const int feature\_type,

DataRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_SET, "value\_type");

int32\_t length = readInt32();

int64\_t id, code;

#ifdef USE\_DENSE\_HASH

record->m\_sparsebinary.resize(2 \* length);

#else

record->m\_sparsebinary.reserve(2 \* length);

#endif

for (int32\_t i = 0; i < length; i++) {

id = readInt64();

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRING, "set:key\_type");

int32\_t set\_length = readInt32();

if (keepKey(id, code)) {

record->m\_sparsebinary[id].reserve(set\_length);

for (int32\_t j = 0; j < set\_length; j++) {

const uint8\_t \*begin = nullptr;

int32\_t str\_len = getRawBuffer<uint8\_t>(&begin);

record->m\_sparsebinary[id].push\_back(bufferToString(str\_len, begin));

}

} else {

for (int32\_t j = 0; j < set\_length; j++) {

int32\_t str\_len = readInt32();

skipLength(str\_len);

}

}

}

}

void DataRecordReader::readSparseContinuous(

const int feature\_type,

DataRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_MAP, "value\_type");

int32\_t length = readInt32();

int64\_t id, code;

#ifdef USE\_DENSE\_HASH

record->m\_sparsecontinuous.resize(2 \* length);

#else

record->m\_sparsecontinuous.reserve(2 \* length);

#endif

for (int32\_t i = 0; i < length; i++) {

id = readInt64();

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRING, "map::key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_DOUBLE, "map::value\_type");

int32\_t map\_length = readInt32();

if (keepKey(id, code)) {

record->m\_sparsecontinuous[id].reserve(map\_length);

for (int32\_t j = 0; j < map\_length; j++) {

const uint8\_t \*begin = nullptr;

int32\_t str\_len = getRawBuffer<uint8\_t>(&begin);

double val = readDouble();

if (!std::isnan(val)) {

record->m\_sparsecontinuous[id].push\_back({bufferToString(str\_len, begin), val});

}

}

} else {

for (int32\_t j = 0; j < map\_length; j++) {

int32\_t str\_len = readInt32();

skipLength(str\_len);

skip<double>();

}

}

}

}

void DataRecordReader::readBlob(

const int feature\_type,

DataRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRING, "value\_type");

int32\_t length = readInt32();

int64\_t id, code;

for (int32\_t i = 0; i < length; i++) {

id = readInt64();

if (keepKey(id, code)) {

const uint8\_t \*begin = nullptr;

int32\_t blob\_len = getRawBuffer<uint8\_t>(&begin);

record->m\_blob[id] = std::vector<uint8\_t>(begin, begin + blob\_len);

} else {

int32\_t str\_len = readInt32();

skipLength(str\_len);

}

}

}

} // namespace twml