package com.twitter.ann.common

import com.google.common.io.ByteStreams

import com.twitter.ann.common.thriftscala.AnnIndexMetadata

import com.twitter.mediaservices.commons.codec.ArrayByteBufferCodec

import com.twitter.mediaservices.commons.codec.ThriftByteBufferCodec

import com.twitter.search.common.file.AbstractFile

import java.io.IOException

import java.io.InputStream

import java.io.OutputStream

import java.nio.channels.Channels

import org.apache.beam.sdk.io.FileSystems

import org.apache.beam.sdk.io.fs.MoveOptions

import org.apache.beam.sdk.io.fs.ResolveOptions

import org.apache.beam.sdk.io.fs.ResolveOptions.StandardResolveOptions

import org.apache.beam.sdk.io.fs.ResourceId

import org.apache.beam.sdk.util.MimeTypes

import org.apache.hadoop.io.IOUtils

import scala.collection.JavaConverters.\_

/\*\*

\* This class creates a wrapper around GCS filesystem and HDFS filesystem for the index

\* generation job. It implements the basic methods required by the index generation job and hides

\* the logic around handling HDFS vs GCS.

\*/

class IndexOutputFile(val abstractFile: AbstractFile, val resourceId: ResourceId) {

// Success file name

private val SUCCESS\_FILE = "\_SUCCESS"

private val INDEX\_METADATA\_FILE = "ANN\_INDEX\_METADATA"

private val MetadataCodec = new ThriftByteBufferCodec[AnnIndexMetadata](AnnIndexMetadata)

/\*\*

\* Constructor for ResourceId. This is used for GCS filesystem

\* @param resourceId

\*/

def this(resourceId: ResourceId) = {

this(null, resourceId)

}

/\*\*

\* Constructor for AbstractFile. This is used for HDFS and local filesystem

\* @param abstractFile

\*/

def this(abstractFile: AbstractFile) = {

this(abstractFile, null)

}

/\*\*

\* Returns true if this instance is around an AbstractFile.

\* @return

\*/

def isAbstractFile(): Boolean = {

abstractFile != null

}

/\*\*

\* Creates a \_SUCCESS file in the current directory.

\*/

def createSuccessFile(): Unit = {

if (isAbstractFile()) {

abstractFile.createSuccessFile()

} else {

val successFile =

resourceId.resolve(SUCCESS\_FILE, ResolveOptions.StandardResolveOptions.RESOLVE\_FILE)

val successWriterChannel = FileSystems.create(successFile, MimeTypes.BINARY)

successWriterChannel.close()

}

}

/\*\*

\* Returns whether the current instance represents a directory

\* @return True if the current instance is a directory

\*/

def isDirectory(): Boolean = {

if (isAbstractFile()) {

abstractFile.isDirectory

} else {

resourceId.isDirectory

}

}

/\*\*

\* Return the current path of the file represented by the current instance

\* @return The path string of the file/directory

\*/

def getPath(): String = {

if (isAbstractFile()) {

abstractFile.getPath.toString

} else {

if (resourceId.isDirectory) {

resourceId.getCurrentDirectory.toString

} else {

resourceId.getCurrentDirectory.toString + resourceId.getFilename

}

}

}

/\*\*

\* Creates a new file @param fileName in the current directory.

\* @param fileName

\* @return A new file inside the current directory

\*/

def createFile(fileName: String): IndexOutputFile = {

if (isAbstractFile()) {

// AbstractFile treats files and directories the same way. Hence, not checking for directory

// here.

new IndexOutputFile(abstractFile.getChild(fileName))

} else {

if (!resourceId.isDirectory) {

// If this is not a directory, throw exception.

throw new IllegalArgumentException(getPath() + " is not a directory.")

}

new IndexOutputFile(

resourceId.resolve(fileName, ResolveOptions.StandardResolveOptions.RESOLVE\_FILE))

}

}

/\*\*

\* Creates a new directory @param directoryName in the current directory.

\* @param directoryName

\* @return A new directory inside the current directory

\*/

def createDirectory(directoryName: String): IndexOutputFile = {

if (isAbstractFile()) {

// AbstractFile treats files and directories the same way. Hence, not checking for directory

// here.

val dir = abstractFile.getChild(directoryName)

dir.mkdirs()

new IndexOutputFile(dir)

} else {

if (!resourceId.isDirectory) {

// If this is not a directory, throw exception.

throw new IllegalArgumentException(getPath() + " is not a directory.")

}

val newResourceId =

resourceId.resolve(directoryName, ResolveOptions.StandardResolveOptions.RESOLVE\_DIRECTORY)

// Create a tmp file and delete in order to trigger directory creation

val tmpFile =

newResourceId.resolve("tmp", ResolveOptions.StandardResolveOptions.RESOLVE\_FILE)

val tmpWriterChannel = FileSystems.create(tmpFile, MimeTypes.BINARY)

tmpWriterChannel.close()

FileSystems.delete(List(tmpFile).asJava, MoveOptions.StandardMoveOptions.IGNORE\_MISSING\_FILES)

new IndexOutputFile(newResourceId)

}

}

def getChild(fileName: String, isDirectory: Boolean = false): IndexOutputFile = {

if (isAbstractFile()) {

new IndexOutputFile(abstractFile.getChild(fileName))

} else {

val resolveOption = if (isDirectory) {

StandardResolveOptions.RESOLVE\_DIRECTORY

} else {

StandardResolveOptions.RESOLVE\_FILE

}

new IndexOutputFile(resourceId.resolve(fileName, resolveOption))

}

}

/\*\*

\* Returns an OutputStream for the underlying file.

\* Note: Close the OutputStream after writing

\* @return

\*/

def getOutputStream(): OutputStream = {

if (isAbstractFile()) {

abstractFile.getByteSink.openStream()

} else {

if (resourceId.isDirectory) {

// If this is a directory, throw exception.

throw new IllegalArgumentException(getPath() + " is a directory.")

}

val writerChannel = FileSystems.create(resourceId, MimeTypes.BINARY)

Channels.newOutputStream(writerChannel)

}

}

/\*\*

\* Returns an InputStream for the underlying file.

\* Note: Close the InputStream after reading

\* @return

\*/

def getInputStream(): InputStream = {

if (isAbstractFile()) {

abstractFile.getByteSource.openStream()

} else {

if (resourceId.isDirectory) {

// If this is a directory, throw exception.

throw new IllegalArgumentException(getPath() + " is a directory.")

}

val readChannel = FileSystems.open(resourceId)

Channels.newInputStream(readChannel)

}

}

/\*\*

\* Copies content from the srcIn into the current file.

\* @param srcIn

\*/

def copyFrom(srcIn: InputStream): Unit = {

val out = getOutputStream()

try {

IOUtils.copyBytes(srcIn, out, 4096)

out.close()

} catch {

case ex: IOException =>

IOUtils.closeStream(out);

throw ex;

}

}

def writeIndexMetadata(annIndexMetadata: AnnIndexMetadata): Unit = {

val out = createFile(INDEX\_METADATA\_FILE).getOutputStream()

val bytes = ArrayByteBufferCodec.decode(MetadataCodec.encode(annIndexMetadata))

out.write(bytes)

out.close()

}

def loadIndexMetadata(): AnnIndexMetadata = {

val in = ByteStreams.toByteArray(getInputStream())

MetadataCodec.decode(ArrayByteBufferCodec.encode(in))

}

}