

In the cloud environment, there is a probability that the dictionary metadata file will be read abnormally during building job, resulting in incorrect query results.

Root Cause

First, let's look at the construction process of the v2 dictionary.

1. First, the number of buckets will be calculated based on the number of data rows, etc. Bucket ≥ 1 . Here, it is assumed that the calculated result is 2.

```
@throws[IOException]
private[builder] def safeBuild(ref: TblColRef): Unit = {
  val sourceColumn = ref.getIdentity
  tryZKJaasConfiguration()
  val lock: Lock = KylinConfig.getInstanceFromEnv.getDistributedLockFactory
    .getLockForCurrentThread(getLockPath(sourceColumn))
  lock.lock()
  try {
    val dictColDistinct = dataset.select(wrapCol(ref)).distinct
    ss.sparkContext.setJobDescription("Calculate bucket size " + ref.getIdentity)
    val bucketPartitionSize = logTime(s"calculating bucket size for $sourceColumn") {
      DictionaryBuilderHelper.calculateBucketSize(seg, ref, dictColDistinct)
    }
    build(ref, bucketPartitionSize, dictColDistinct)
  } finally lock.unlock()
}
```

2. Then use this value to partition the data, and write the dictionary code to the `CURR_{id}` file according to the partition number, that is, how many partitions (`bucketPartitionSize`) will generate how many CURR files, this example assumes that the `bucketPartitionSize` is 2, so two CURR files will be generated, respectively: CURR_0 and CURR_1

```

@throws[IOException]
private[builder] def build(ref: TblColRef, bucketPartitionSize: Int,
    afterDistinct: Dataset[Row]): Unit = logTime( action = s"building global dictionaries V2 for ${ref.getIdentity}") {
    val globalDict = new NGlobalDictionaryV2(seg.getProject, ref.getTable, ref.getName, seg.getConfig.getHdfsWorkingDirectory)
    globalDict.prepareWrite()
    val broadcastDict = ss.sparkContext.broadcast(globalDict)

    changeAQEConfig( isDictBuildFinished = false)
    ss.sparkContext.setJobDescription("Build dict " + ref.getIdentity)

    val dictCol = col(afterDistinct.schema.fields.head.name)
    afterDistinct.filter(dictCol.isNotNull)
    .repartition(bucketPartitionSize, dictCol)
    // https://issues.apache.org/jira/browse/SPARK-32051
    .foreachPartition((iter: Iterator[Row]) => {
        val partitionID = TaskContext.get().partitionId()
        logInfo(msg = s"Build partition dict col: ${ref.getIdentity}, partitionId: $partitionID")
        val broadcastGlobalDict = broadcastDict.value
        val bucketDict = broadcastGlobalDict.loadBucketDictionary(partitionID)
        iter.foreach(r => bucketDict.addRelativeValue(r.getString(0)))

        bucketDict.saveBucketDict(partitionID)
    })
}

```

3. Finally, write the `bucketPartitionSize` into the meta file

```

globalDict.writeMetaDict(bucketPartitionSize, seg.getConfig.getGlobalDictV2MaxVersions, seg.getConfig.getGlobalDictV2VersionTTL)

```

The corresponding class field is the `bucketSize` variable in `NGlobalDictMetaInfo`.

```

NGlobalDictMetaInfo(int bucketSize, long[] bucketOffsets, long dictCount, long[] bucketCount) {
    this.bucketSize = bucketSize;
    this.dictCount = dictCount;
    this.bucketOffsets = bucketOffsets;
    this.bucketCount = bucketCount;
}

long getOffset(int point) { return bucketOffsets[point]; }

public int getBucketSize() {
    return bucketSize;
}

```

Then look at the coding logic of the dictionary column on the flat table

1. First, you need to create an encoding parameter, which involves only one meta file, that is, the number of `bucketSize` obtained through the meta file. The parameter is a 5-tuple, and the last item is false if the `bucketSize` is greater than 1, and true if it is equal to 1;

Key code: here you need to pass `bucketSize` to the `dict_encode` for encoding, and then return the encoded column as a project

```

val encodingArgs = encodingCols.map {
  ref =>
    // val globalDict = new NGlobalDictionaryV2(seg.getProject, ref.getTable, ref.getName, seg.getConfig.getHdfsWorkingDirectory, "niubi")
    val globalDict = new NGlobalDictionaryV2(seg.getProject, ref.getTable, ref.getName, seg.getConfig.getHdfsWorkingDirectory)

    // val bucketSize = globalDict.getBucketSizeOrRecalculate(seg.getConfig.getGlobalDictV2MinHashPartitions)

    val bucketSize = globalDict.getBucketSizeOrDefault(seg.getConfig.getGlobalDictV2MinHashPartitions)
    val enlargedBucketSize = ((minBucketSize / bucketSize) + 1) * bucketSize.toInt
    logInfo(msg = "=====> bucketSize: " + bucketSize)
    val encodeColRef = convertFromDot(ref.getBackTickIdentity)
    val columnIndex = structType.fieldIndex(encodeColRef)

    val dictParams = Array(seg.getProject, ref.getTable, ref.getName, seg.getConfig.getHdfsWorkingDirectory)
      .mkString(SEPARATOR)
    val aliasName = structType.apply(columnIndex).name.concat(ENCODE_SUFFIX)
    val encodeCol = dict_encode(col(encodeColRef).cast(StringType), lit(dictParams), lit(bucketSize).cast(StringType)).as(aliasName)
    // val encodeCol = dict_encode(col(encodeColRef).cast(StringType), lit(dictParams), lit(1).cast(StringType)).as(aliasName)
    val columns = encodeCol
    (enlargedBucketSize, col(encodeColRef).cast(StringType), columns, aliasName,
     bucketSize == 1)
}

encodingArgs.foreach {
  case (enlargedBucketSize, repartitionColumn, projects, _ false) =>
    partitionedDs = partitionedDs
      .repartition(enlargedBucketSize, repartitionColumn)
      .select(partitionedDs.schema.map(ty => col(ty.name)) ++ Seq(projects): _*)
  case (_, _, projects, _ true) =>
    partitionedDs = partitionedDs
      .select(partitionedDs.schema.map(ty => col(ty.name)) ++ Seq(projects): _*)
}
ds.sparkSession.sparkContext.setJobDescription(null)
partitionedDs
}
}

```

- The logic of getting the number of `bucketSize` through the meta file is as follows. If the meta file is not read, return a default value of 1. If the meta file is read, return the value recorded in the meta file

Although the meta file was not read, due to the default, no errors were reported and encoding could continue.

```

public int getBucketSizeOrDefault(int defaultSize) {
    int bucketPartitionSize;
    if (metadata == null) {
        bucketPartitionSize = defaultSize;
    } else {
        bucketPartitionSize = metadata.getBucketSize();
    }

    return bucketPartitionSize;
}

```

- Finally, use the created encoding parameters to encode, that is, use the encoded project to append to the original table

```

val encodingArgs = encodingCols.map {
  ref =>
    // val globalDict = new NGlobalDictionaryV2(seg.getProject, ref.getTable, ref.getName, seg.getConfig.getHdfsWorkingDirectory, "niubi")
    val globalDict = new NGlobalDictionaryV2(seg.getProject, ref.getTable, ref.getName, seg.getConfig.getHdfsWorkingDirectory)

    // val bucketSize = globalDict.getBucketSizeOrRecalculate(seg.getConfig.getGlobalDictV2MinHashPartitions)

    val bucketSize = globalDict.getBucketSizeOrDefault(seg.getConfig.getGlobalDictV2MinHashPartitions)
    val enlargedBucketSize = ((minBucketSize / bucketSize) + 1) * bucketSize.toInt
    logInfo(msg = "=====> bucketSize: " + bucketSize)
    val encodeColRef = convertFromDot(ref.getBackTickIdentity)
    val columnIndex = structType.fieldIndex(encodeColRef)

    val dictParams = Array(seg.getProject, ref.getTable, ref.getName, seg.getConfig.getHdfsWorkingDirectory)
      .mkString(SEPARATOR)
    val aliasName = structType.apply(columnIndex).name.concat(ENCODE_SUFFIX)
    val encodeCol = dict_encode(col(encodeColRef).cast(StringType), lit(dictParams), lit(bucketSize).cast(StringType)).as(aliasName)
    // val encodeCol = dict_encode(col(encodeColRef).cast(StringType), lit(dictParams), lit(1).cast(StringType)).as(aliasName)
    val columns = encodeCol
    (enlargedBucketSize, col(encodeColRef).cast(StringType), columns, aliasName,
     bucketSize == 1)
}

encodingArgs.foreach {
  case (enlargedBucketSize, repartitionColumn, projects, _) => false =>
    partitionedDs = partitionedDs
      .repartition(enlargedBucketSize, repartitionColumn)
      .select(partitionedDs.schema.map(ty => col(ty.name)) ++ Seq(projects): _*)
  case (_, _, projects, _) => true =>
    partitionedDs = partitionedDs
      .select(partitionedDs.schema.map(ty => col(ty.name)) ++ Seq(projects): _*)
}
ds.sparkSession.sparkContext.setJobDescription(null)
partitionedDs
}
}

```

So the problem is, at the beginning of this example, it has been assumed that the bucket for building the dictionary is 2, but since the meta file was not read during the encoding stage, the bucketSize is the default value of 1, so the bucketSize passed to the dict_encode is also 1, and finally Only one curr file is read, so the data of another curr file is lost, so the final encoded result is also missing

In a word, the reason for the error is that the number of partitions calculated when building the dictionary is inconsistent with the number of partitions read from the meta file when encoding the flat table, resulting in incomplete flat table encoding and inconsistent query results.

Fix Design:

If find that the meta file or curr file does not exist, report an error directly and throw an exception: `FileNotFoundException ("GLOBAL DICT META FILE NOT FOUND, ENCODETABLE FAILED");` and print the log, output all filenames in the current version directory