Chapter 4 TFRecord Implementation

Though TFRecord is a highly developed ecosystem, it doesn't support full-range random access, neither is full-range random shuffling. Due to the previously explained pros, we think it's good to enable TFRecord to be full-range random shuffled, by applying LIRS to it. First, we made TFRecord random-accessible. Programmers can also use padding for TFRecord with our implementation. After realizing random-accessibility of TFRecord file, we then apply LIRS mechanism to it, including Page-Aware Random Shuffling.

4.1 Apply LIRS to TFRecord

In order to fully shuffle a TFRecord file, we need to make TFRecord random-accessible. Several functions are necessary. We implemented four new functions and modified two existing functions. First, **TFRecordLIRS.read(instance id)** returns the corresponding instance of the specified instance id. This function is the backbone of random-accessing TFRecord, therefore is also used inside the following random iterator. Second, for ease of use, it needs to have a random iterator **TFRecordLIRS.next()**, which returns a random instance again and again when being called. Third, random access to an arbitrary instance cannot be done unless the offset of a given instance is known if the dataset is sparse. We need to build a corresponded offset_table when building a TFRecord file. Fourth, to use

LIRS, we need **TFRecordLIRS.shuffle_order(page_aware)** to generate a random assign table. Finally, we need to write out the offset_table and the TFRecord file by **TFRecordWriter.write()**. The implementation details to enable random access to TFRecords are explained in the following.

1. **TFRecordLIRS.read(instance id)**: This is a new module. It takes a specified instance id as input. It then searches in the offset_table and gets the length of the specified instance by Equation 4.1. Finally, it reads and returns the instance. For applying LIRS, programmers don't need to directly use this function as **TFRecordLIRS.next()** is a higher level abstraction. However, programmers can use this function to random access a TFRecord file.

$$instance\ length = next\ instance\ offset\ - current\ instance\ offset \quad (4.1)$$

2. **TFRecordLIRS.next()**: This is a modified module. It should be used after using **TFRecordLIRS.shuffle_order(page_aware)**. Instead of returning the instance on the top of the TFIP, it returns instances from the TFRecord file as the order **TFRecordLIRS.shuffle_order(page_aware)** generates.

3. **OffsetTableBuilder.calc_offset(instance)**: This is a new module. It uses Equation 4.2 to calculate the offset of each instance and store it inside an array. The maximum size of this array is shown in Equation 4.3. We also need to decide whether to use the padding mechanism when initializing an OffsetTableBuilder object. If so, the equation calculating instance offsets would be replaced by Algorithm 1 or Algorithm 2, which are shown in the previous section. An example will be shown

in the following section.

$$current\ instance\ offset = previous\ instance\ offset + previous\ instance\ length \tag{4.2}$$

$$offset\_table\_size = \#\ of\ instances\ *\ 8\ Bytes \tag{4.3}$$

4. **OffsetTableBuilder.write(offset_table_filename)**: This is a new module. It writes the offset table built by **OffsetTableBuilder.calc_offset()** to an offset table file.

5. **TFRecordLIRS.shuffle_order(page_aware)**: This is a new module. It (re-)shuffles the random assign table for the TFRecord file. Whether to use page-aware random shuffling is also specified in this module.

6. **TFRecordWriter.write(instance or padding)**: This is a modified module. It writes an instance or padding to the TFRecord file.

At the end, we succeeded to apply LIRS to TFRecord. The code can be found on Github: https://github.com/winiel559/tfrecord-lirs-implementation.

4.2 Example of Applying LIRS to TFRecord

The followings are examples briefly showing how to apply LIRS to TFRecord with our implementation. The first part is an example of how to build the TFRecord file which supports random access.

```
1 input: instances[], whether_to_use_padding
```

```
output: offset_table_file, TFRecord_file

offset_table=OffsetTableBuilder(whether_to_use_padding)

for all instance in instances[]:
    # ...
    # Do whatever pre-processing you want, e.g. rezising
      image, filtering low quality data
    # ...

    if whether_to_use_padding is True:
        # write the padding to TFRecord_file
        TFRecordWriter.write(padding)
    # calc the offset of this instance
    OffsetTableBuilder.calc_offset(instance)
    # write the instance to TFRecord_file
    TFRecordWriter.write(instance)

OffsetTableBuilder.write(offset_table_file)
```

The second part is an example of how to use the built TFRecord file and offset_table to train the model using LIRS. Note that whether to use padding is decided when building the TFRecord file. As the offset table tells TFRecord-LIRS where each instance is, we don't need to know if the dataset is padded when training.

```
input: offset_table_file, TFRecord_file, page_aware(
    whether_to_use_page-aware-assignment)
output: the trained model

# Initialize the TFRecord-LIRS object, this initialization
    also opens the TFRecord file and the offset table file.
train_dataset = TFRecordLIRS(TFRecord_file,
    offset_table_file)

for number of epochs:
    train_dataset.shuffle_order(page_aware)
    for number of steps:
        batch = []
        for batchsize:
            # form a batch by calling next() to
            batch.append(train_dataset.next())
        # note that this train step is executed by GPU
        train(model, batch)
```