Bucket Id range increase proposal

Problem statement

The bucket id in BucketCodec.V1 is stored in a twelve bit part of a 32bit integer.

- top 3 bits - version code.
- next 1 bit - reserved for future
- **next 12 bits - the bucket ID**
- next 4 bits reserved for future
- remaining 12 bits - the statement ID - 0-based numbering of all statements within a

Its maximum value is 4095.

Depending on TEZ parallelism settings more than 4095 tasks might be started and each of those are supposed to write its own bucket.

On the high level taskId corresponds to bucketId and FS_OPERATOR_ID corresponds to statement id.

If more than 4095 tasks are started the bucket id overflows and the operation fails with the following exception:

```
org.apache.hadoop.hive.ql.exec.tez.MapRecordSource.processRow(MapRecordSource.java:92) ... 18 more Caused by: org.apache.hadoop.hive.ql.metadata.HiveException: java.lang.IllegalArgumentException: bucketId out of range: 4098 at org.apache.hadoop.hive.ql.exec.FileSinkOperator.createBucketFiles(FileSinkOperator.java:820) at org.apache.hadoop.hive.ql.exec.FileSinkOperator.process(FileSinkOperator.java:995) at org.apache.hadoop.hive.ql.exec.Operator.forward(Operator.java:938) at org.apache.hadoop.hive.ql.exec.SelectOperator.process(SelectOperator.java:95) at org.apache.hadoop.hive.ql.exec.Operator.forward(Operator.java:938) at org.apache.hadoop.hive.ql.exec.TableScanOperator.process(TableScanOperator.java:174) at org.apache.hadoop.hive.ql.exec.MapOperator$MapOpCtx.forward(MapOperator.java:152) at org.apache.hadoop.hive.ql.exec.MapOperator.process(MapOperator.java:552) ... 19 more Caused by: java.lang.IllegalArgumentException: bucketId out of range: 4098 at org.apache.hadoop.hive.ql.io.BucketCodec$2.encode(BucketCodec.java:94) at org.apache.hadoop.hive.ql.io.orc.OrcRecordUpdater.<init>(OrcRecordUpdater.java:270) at org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat.getRecordUpdater(OrcOutputFormat.java:289) at org.apache.hadoop.hive.ql.io.HiveFileFormatUtils.getRecordUpdater(HiveFileFormatUtils.java:352) at org.apache.hadoop.hive.ql.io.HiveFileFormatUtils.getAcidRecordUpdater(HiveFileFormatUtils.java:338) at org.apache.hadoop.hive.ql.exec.FileSinkOperator.createBucketForFileIdx(FileSinkOperator.java:883) at org.apache.hadoop.hive.ql.exec.FileSinkOperator.createBucketFiles(FileSinkOperator.java:814) ... 26 more ],
```

Workarounds

One workaround is to tweak **tez.grouping.min-size** size so that the split count after grouping is less than 4096.

The problem with that is that there are no clear instructions on how to set this, it depends on the data characteristics.

Customers keep facing this problem and they need to work around it by trial and error.

Proposed solution

The proposed solution is to let the bucket id overflow into the statement id. Depending on the maximum statement id we can dynamically allocate bits for the bucket id from the statement id field.

For example in a single statement operation only one bit is needed for the statement id and 11 extra bits can be allocated for the bucket id.

In case of a multi insert where the max statement id lets say 100 (0b000001100100), five extra bits can be allocated for the bucket Id.

Example:

Bucket Id = 5000
Statement Id = 50
Max statement Id = 100 (number of file sinks)

```
Bucket Id:              1001110001000 (1 bit overflows)
Max statement Id:        000001100100 (5 bit free)
Statement Id:            000000110010

New bucket id:           001110001000 (truncated)
New statement Id:        000010110010 (one bit is taken out of 5)
```

5000 buckeId becomes 904 (5000 modulo 4096) and it looks like (intentionally, see note on compaction) it was created by stmt = 178.

Therefore 5000 and 904 are both the same buckets on ORC level but they’re in different delta directories.

1. delta_1_1_0/bucket_00904_0
2. delta_1_1_178/bucket_00904_0

POC implementation

```
if (bucketId > MAX_BUCKET_ID) {
 int extraBits = NUM_STATEMENT_ID_BITS - (32 -
Integer.numberOfLeadingZeros(maxStatementId));

 int overflowedParts = bucketId >>> NUM_BUCKET_ID_BITS;
 int maxBucketId = (1 << (NUM_BUCKET_ID_BITS + extraBits)) -1;
 Preconditions.checkArgument(bucketId >= 0 && bucketId <= maxBucketId,
"Bucket ID out of range: " + bucketId + " max: " + maxBucketId);

 statementId = (overflowedParts << (NUM_STATEMENT_ID_BITS - extraBits)) |
statementId;
 bucketId = bucketId & MAX_BUCKET_ID;
}
```

```
int maxStmtId = -1;
for (FileSinkDesc acidSink : acidSinks) {
  TableDesc tableInfo = acidSink.getTableInfo();
  TableName tableName = HiveTableName.of(tableInfo.getTableName());
  long writeId = driverContext.getTxnManager().getTableWriteId(tableName.getDb(), tableName.getTable());
  acidSink.setTableWriteId(writeId);

  /**
   * it's possible to have > 1 FileSink writing to the same table/partition
   * e.g. Merge stmt, multi-insert stmt when mixing DP and SP writes
   * Insert ... Select ... Union All Select ... using
   * {@link org.apache.hadoop.hive.ql.exec.AbstractFileMergeOperator#UNION_SUDBIR_PREFIX}
   */
  acidSink.setStatementId(driverContext.getTxnManager().getStmtIdAndIncrement());
  maxStmtId = Math.max(acidSink.getStatementId(), maxStmtId);
  String unionAllSubdir = "/" + AbstractFileMergeOperator.UNION_SUDBIR_PREFIX;
  if (acidSink.getInsertOverwrite() && acidSink.getDirName().toString().contains(unionAllSubdir) &&
      acidSink.isFullAcidTable()) {
    throw new UnsupportedOperationException("QueryId=" + driverContext.getPlan().getQueryId() +
        " is not supported due to OVERWRITE and UNION ALL.  Please use truncate + insert");
  }
}
for (FileSinkDesc each : acidSinks) {
  each.setMaxStmtId(maxStmtId);
}
```

After running an INSERT with 6000 parallel tasks we end up having 2 delta directories:

- `/warehouse/tablespace/managed/hive/cdr5bi/delta_0000039_0000039_0000`
- `/warehouse/tablespace/managed/hive/cdr5bi/delta_0000039_0000039_0002`

Where in the first we have buckets from 0 .. 4095 and in the second we have buckets from 0 .. 1903.

After running compaction we expect having a single base directory with buckets from 0 .. 4095.

- `/warehouse/tablespace/managed/hive/cdr5bi/base_0000039_v0003975`

The same buckets from different statements are merged together.

Advantages

The main advantage is that we don't limit parallelism settings. We can start 5000 tasks in this example and each of those will write their own buckets.

The solution is supposed to be backward compatible with the existing implementation (unlike increasing the bucket id range by using 64 bit integer instead of 32).

One assumption is that compaction ignores statement ids therefore the same buckets (delta_1_1_0/bucket_00904_0 and delta_1_1_178/bucket_00904_0) will be merged together after running a major compaction, so we won't end up having more than 4096 files in the end.

Simply increasing the range would have an undesirable effect of making compaction less efficient.

Found problems

In FileSink>>jobClose() we process the commit manifest files which are located under a temp folder per statement id.

1. _tmp.delta_0000050_0000050_0000 // statementId = 0
2. _tmp.delta_0000050_0000050_0002 // statementId = 2

However the jobClose is running on HiveServer2 and this FileSink is the base one that is populated with compile time information it doesn't have any knowledge on how many times the

bucketId overflowed and how many new statement id was introduced. So it only deals with its own folder (statementId = 0).

The end results are some unprocessed manifest files and an undeleted _tmp folder.

This can be solved by always using FS_OP_ID as statement id when generating the manifest directories. However the logic that validates the manifests at the end should be changed.

The easiest might be to iterate over the manifest files and collect their parent directories into a set and use that as directInsertDirectories.

Full POC implementation: https://github.com/apache/hive/pull/1968/