BP-6 - Use separate log for compaction

Problem

Bookkeeper is not able to reclaim disk space when it's full

If all disks are full or almost full, both major and minor compactions would be suspended, and only GC will be running. In the current design, this is the right thing to do, because when disks are full, EntryLogger can not allocate any new entry logs any more, and apart from that, the intention is to prevent disk usage from keep growing.

However, the problem is if we have a mixed of short-lived ledgers and long-lived ledgers in all entry logs, when disks are full, GC wouldn't be able to delete any entry logs, and if compaction is disabled, bookie can't reclaim any disk space any more by itself.

Compaction might keep generating duplicated data which would cause disk full

Currently, there's no transactional operation for compaction. In the current CompactionScannerFactory, if it fails to flush entry log file, or fails to flush ledgerCache, the data which is already flushed wouldn't be deleted, and the entry log that is being compacted will be retried again for the next time, which would generate duplicated data.

Moreover, if the entry log being compacted has long-lived data and the compaction keeps failing for some reason(e.g. corrupted entry, corrupted index), it would cause the BK disk usage keep growing until the either the entry log can be garbage collected, or disk full.

Proposal

Use a separate log for compaction

In order to address the first issue, we can use a separate log file for compaction and have a separate allocation logic. To allocate a compaction log file, we don't have to choose from the writable ledger directories, which is determined by diskWarnThreshold. In fact, as long as there's a ledger directory that has enough disk space for the next compaction log (we can use log size limit), we should be good to allocate. Because when disks are full, bookie must be running in read only mode, only compaction would write to ledger disks.

Add transactional phases for compaction

Once we separate the log file for compaction, we can achieve a transactional compaction operation. By "transactional", we mean that if anything fail at any phases during compaction, we should be able to roll back the current compaction properly, failed compaction would still be able to retry in the next scan, but rolling back the failed compaction would help us clean up the duplicated data.

Add recovery for compaction

If for any reason we succeed in flushing the compaction log but then fail to update the index or flush the ledger cache, we would get into the situation that the index is partially updated. In this case, we can't simply roll back by deleting the old entry log or compaction log, because the partially updated index file might already pointing some ledgers to the new compaction log. So we need a way to recover the partially updated index file for the compaction log file.

Design

Create a separate file for compaction but share the same allocation logic

The idea is that we reuse the current allocation logic but use a different file suffix for compaction. In this way, we can make minimum changes but achieve the goal of using a separate file for compaction.

```
EntryLogWriteChannel createNewCompactionLog() throws IOException {
    synchronized (createCompactionLogLock) {
        List<File> writableDirs;
        try {
            writableDirs = new ArrayList<>(ledgerDirsManager.getWritableLedgerDirs());
        } catch (LedgerDirsManager.NoWritableLedgerDirException nlde) {
            // if all ledgers dirs are full, try to pick a ledger dir that has enough
space for compaction log.
            writableDirs = ledgerDirsManager.getAllLedgerDirs();
            for (File dir : writableDirs) {
                if (dir.getUsableSpace() <= logSizeLimit) {
                    writableDirs.remove(dir);
                }
            }
        }
        EntryLogWriteChannel bc = allocateNewLogFile(COMPACTING_SUFFIX, writableDirs);
        LOG.info("Created new compaction logger {}.", bc.getLogId());
        return bc;
    }
}
```

```
/**
 * Allocate a new entry log file
 */
EntryLogWriteChannel allocateNewLogFile(String suffix, List<File> writableDirs)
    throws IOException {
    if (writableDirs.isEmpty()) {
        throw new LedgerDirsManager.NoWritableLedgerDirException(
            "No writable ledger directories to allocate new entry log.");
    }
    Collections.shuffle(writableDirs);
    synchronized (this) {
        // It would better not to overwrite existing entry log files
        File newLogFile = null;
        do {
            if (nextLogId >= Integer.MAX_VALUE) {
                nextLogId = 0;
            } else {
                ++nextLogId;
            }
            String logFileName = Long.toHexString(nextLogId) + suffix;
            for (File dir : writableDirs) {
                newLogFile = new File(dir, logFileName);
                if (newLogFile.exists()) {
                    LOG.warn("Found existed entry log " + newLogFile
                        + " when trying to create it as a new log.");
                    newLogFile = null;
                    break;
                }
            }
        } while (newLogFile == null);

        FileChannel channel = new RandomAccessFile(newLogFile, "rw").getChannel();
        EntryLogWriteChannel logChannel = new EntryLogWriteChannel(nextLogId, channel,
            newLogFile, serverCfg.getWriteBufferBytes(),
serverCfg.getReadBufferBytes());
        logChannel.writeHeader((ByteBuffer) LOGFILE_HEADER.clear());

        for (File dir : writableDirs) {
            try {
                setLastLogId(dir, nextLogId);
            } catch (IOException ioe) {
                LOG.warn("Failed to write lastId {} to directory {} : ",
                    new Object[]{nextLogId, dir, ioe});
            }
        }
        LOG.info("Preallocated log file {} for logId {}.", newLogFile, nextLogId);
        return logChannel;
    }
}
```

Introduce Compaction Transactional Phases

We can add a separate class CompactionWorker to handle all the compaction logic in the same place. The main compact function would be like this:

```
/**
 * Compaction is composed of 3 transactional phases:
 * 1. Scan data from entry log file into compaction log file and keep track of new
entry locations
 * 2. Flush compaction log to make sure it's persisted in disk and roll a new
compaction log
 * 3. Update entry locations to cache and flush index, finally delete the old entry
log file
 * <p>
 * If compaction failed in any phase, abort the current compaction gracefully.
 */
synchronized boolean compact(EntryLogMetadata metadata) {
    if (metadata != null) {
        LOG.info("Compacting entry log {} : {}.", metadata.entryLogId, metadata);
        CompactionPhase scanEntryLog = new ScanEntryLogPhase(metadata);
        if (!scanEntryLog.run()) {
            LOG.info("Compaction for entry log {} end in ScanEntryLogPhase.",
metadata.entryLogId);
            return false;
        }
        File compactionLogFile = entryLogger.getCurCompactionLogFile();
        CompactionPhase flushCompactionLog = new
FlushCompactionLogPhase(metadata.entryLogId);
        if (!flushCompactionLog.run()) {
            LOG.info("Compaction for entry log {} end in FlushCompactionLogPhase.",
metadata.entryLogId);
            return false;
        }
        File compactedLogFile = getCompactedLogFile(compactionLogFile,
metadata.entryLogId);
        CompactionPhase updateIndex = new UpdateIndexPhase(compactedLogFile);
        if (!updateIndex.run()) {
            LOG.info("Compaction for entry log {} end in UpdateIndexPhase.",
metadata.entryLogId);
            return false;
        }
        LOG.info("Compacted entry log : {}.", metadata.entryLogId);
        return true;
    }
    return false;
}
```

And this is the abstract class for all compaction phases.

```
/**
 * An abstract class that would be extended to be the actual transactional phases for
compaction
 */
abstract static class CompactionPhase {
    private String phaseName = "";

    CompactionPhase() {
    }

    CompactionPhase(String phaseName) {
        this.phaseName = phaseName;
    }

    boolean run() {
        try {
            start();
            return complete();
        } catch (IOException e) {
            LOG.error("Encounter exception in compaction phase {}. Abort current
compaction.", phaseName, e);
            abort();
        }
        return false;
    }

    abstract void start() throws IOException;

    abstract boolean complete() throws IOException;

    abstract void abort();

}
```

```
/**
 * Assume we're compacting entry log 1 to entry log 3.
 * The first phase is to scan entries in 1.log and copy them to compaction log file
"3.log.compacting".
 * We'll try to allocate a new compaction log before scanning to make sure we have a
log file to write.
 * If after scanning, there's no data written, it means there's no valid entries to be
compacted,
 * so we can remove 1.log directly, clear the entry locations and end the compaction.
 * Otherwise, we should move on to the next phase.
 *
 * If anything failed in this phase, we should delete the compaction log and clean the
offsets.
 */
class ScanEntryLogPhase extends CompactionPhase {
 ...
}
```

```
/**
 * Assume we're compacting log 1 to log 3.
 * This phase is to flush the compaction log.
 * When this phase starts, there should be a compaction log file like
"3.log.compacting"
 * When compaction log is flushed, in order to indicate this phase is completed,
 * a hardlink file "3.log.1.compacted" should be created, and "3.log.compacting"
should be deleted.
 */
class FlushCompactionLogPhase extends CompactionPhase {
 ...
}
```

```
/**
 * Assume we're compacting log 1 to log 3.
 * This phase is to update the entry locations and flush the index.
 * When the phase start, there should be a compacted file like "3.log.1.compacted",
 * where 3 is the new compaction logId and 1 is the old entry logId.
 * After the index the flushed successfully, a hardlink "3.log" file should be
created,
 * and 3.log.1.compacted file should be deleted to indicate the phase is succeed.
 *
 * This phase can also used to recover partially flushed index when we pass
isInRecovery=true
 */
class UpdateIndexPhase extends CompactionPhase {
 ...
}
```