The basics of file systems

Neapolis University Pafos

JET BRAINS

What we will implement

Suppose you implement a procedure to merge components of an LSM tree. This procedure needs to store the result in GCS:

1. The size of the result is not known up-front. Moreover, the result may be too big to fit in RAM.
2. The connection to GCS is unreliable. However, the merge procedure itself must not implement retries.

What we will implement

Suppose you implement a procedure to merge components of an LSM tree. This procedure needs to store the result in GCS:

1. The size of the result is not known up-front. Moreover, the result may be too big to fit in RAM.
2. The connection to GCS is unreliable. However, the merge procedure itself must not implement retries.

Suggested implementation:

```
type ReliableWriter struct {
    ???
}

func (w *ReliableWriter) WriteAt(ctx context.Context, buf []byte, off int64) (err error) {
    // in fact, all writes need to be appends; off is a consistency check
    ???
}

func (w *ReliableWriter) Complete(ctx context.Context) (err error) {
    ???
}

func (w *ReliableWriter) Abort(ctx context.Context) {
    ???
}
```

Reminder: uploading big objects

S3 and Azure	GCS
1. `CreateMultipartUpload()`, 2. `PUT Object` to upload parts, possibly in parallel, 3. `CompleteMultipartUpload()`.	1. `POST /upload` to start a resumable upload session, 2. Issue a sequence of PUTs that specify Range: PUTs can't run in parallel, ranges must be adjacent. 3. The last PUT has a special flag that completes an upload.

What we will implement

Suppose you implement a procedure to merge components of an LSM tree. This procedure needs to store the result in GCS:

1. The size of the result is not known up-front. Moreover, the result may be too big to fit in RAM.
2. The connection to GCS is unreliable. However, the merge procedure itself must not implement retries.

Suggested implementation:

```
type ReliableWriter struct {…}

func (w *ReliableWriter) WriteAt(ctx context.Context, buf []byte, off int64) (err error) {…}
func (w *ReliableWriter) Complete(ctx context.Context) (err error) {…}
func (w *ReliableWriter) Abort(ctx context.Context) {…}
```

Object storages have different APIs for chunked uploads. Our server that uses multiple connections for file transfers will have yet another API.

We need to introduce an adapter to the underlying storage.

What we will implement

Suppose you implement a procedure to merge components of an LSM tree. This procedure needs to store the result in GCS:

1. The size of the result is not known up-front. Moreover, the result may be too big to fit in RAM.
2. The connection to GCS is unreliable. However, the merge procedure itself must not implement retries.

Suggested implementation:

```
type ReliableWriter struct {…}

func (w *ReliableWriter) WriteAt(ctx context.Context, buf []byte, off int64) (err error) {…}
func (w *ReliableWriter) Complete(ctx context.Context) (err error) {…}
func (w *ReliableWriter) Abort(ctx context.Context) {…}
```

Object storages have different APIs for chunked uploads. Our server that uses multiple connections for file transfers will have yet another API.

We need to introduce an adapter to the underlying storage.

Question: where do we implement retries and backoffs? In ReliableWriter or in adapters to storages?

What we will implement

Suppose you implement a procedure to merge components of an LSM tree. This procedure needs to store the result in GCS:

1. The size of the result is not known up-front. Moreover, the result may be too big to fit in RAM.
2. The connection to GCS is unreliable. However, the merge procedure itself must not implement retries.

Suggested implementation:

```
type ReliableWriter struct {…}

func (w *ReliableWriter) WriteAt(ctx context.Context, buf []byte, off int64) (err error) {…}
func (w *ReliableWriter) Complete(ctx context.Context) (err error) {…}
func (w *ReliableWriter) Abort(ctx context.Context) {…}
```

Object storages have different APIs for chunked uploads. Our server that uses multiple connections for file transfers will have yet another API.

We need to introduce an adapter to the underlying storage.

Question: where do we implement retries and backoffs? In ReliableWriter or in adapters to storages?

Question: where do we implement chunking? In the client or in the server? Hint: we want to minimise the server's RAM usage.

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

What a reliable writer needs to do:

1. Collect buffers provided to `WriteAt()` into a scatter-gather buffer.
2. Once there is enough data for a chunk, start a goroutine to write it.
3. If there are too many buffers queued for writing, then throttle the writer.
4. The chunk writer goroutine must retry if a chunk upload fails.

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

What a reliable writer needs to do:

1. Collect buffers provided to `WriteAt()` into a scatter-gather buffer.
2. Once there is enough data for a chunk, start a goroutine to write it.
3. If there are too many buffers queued for writing, then throttle the writer.
4. The chunk writer goroutine must retry if a chunk upload fails.

A scatter-gather buffer is a data structure that implements the following operations efficiently:

1. Append a buffer to the tail.
2. Splice another sgbuf to the tail.
3. Take **n** bytes from the head (the result is an sgbuf).

Reminder: this is very much a pipe buffer. See also

1. man 2 splice
2. man 2 vmsplice
3. man 2 tee
4. man 2 sendfile

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

What a reliable writer needs to do:

1. Collect buffers provided to `WriteAt()` into a scatter-gather buffer.
2. Once there is enough data for a chunk, start a goroutine to write it.
3. If there are too many buffers queued for writing, then throttle the writer.
4. The chunk writer goroutine must retry if a chunk upload fails.

A scatter-gather buffer is a data structure that implements the following operations efficiently:

1. Append a buffer to the tail.
2. Splice another sgbuf to the tail.
3. Take **n** bytes from the head (the result is an sgbuf).

If we assume that `WriteAt()` transfers the ownership of buffers to `ReliableWriter`, we may implement a scatter-gather buffer without `memcpy()`s.

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

What a reliable writer needs to do:

1. Collect buffers provided to `WriteAt()` into a scatter-gather buffer.
2. Once there is enough data for a chunk, start a goroutine to write it.
3. If there are too many buffers queued for writing, then throttle the writer.
4. The chunk writer goroutine must retry if a chunk upload fails.

Question: how do we choose the chunk size? It is important to minimise the RAM usage and introduce no latencies.
Static chunk choice can’t be optimal: 256 kbytes per chunk means too slow upload, 32 mbytes per chunk means that `ReliableWriter` must wait for buffers to accumulate, which introduces latency.

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

What a reliable writer needs to do:

1. Collect buffers provided to `WriteAt()` into a scatter-gather buffer.
2. Once there is enough data for a chunk, start a goroutine to write it.
3. If there are too many buffers queued for writing, then throttle the writer.
4. The chunk writer goroutine must retry if a chunk upload fails.

Question: how do we choose the chunk size? It is important to minimise the RAM usage and introduce no latencies.
Static chunk choice can't be optimal: 256 kbytes per chunk means too slow upload, 32 mbytes per chunk means that `ReliableWriter` must wait for buffers to accumulate, which introduces latency.

Idea: choose chunk sizes adaptively. As soon as there is enough data to write a chunk, start a chunk write and let more buffers queue up while a chunk write is proceeding. Then use all available data for the next chunk. Also, chunk sizes must be capped to limit the number of bytes to resend during a retry. That is important for storages like S3 that can't resume chunk uploads halfway through.

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

What a reliable writer needs to do:

1. Collect buffers provided to `WriteAt()` into a scatter-gather buffer.
2. Once there is enough data for a chunk, start a goroutine to write it.
3. If there are too many buffers queued for writing, then throttle the writer.
4. The chunk writer goroutine must retry if a chunk upload fails.

The user of `ReliableWriter` may produce data faster than it can be written. When the total size of buffers queued for writing goes above a (configurable) limit, a call to `WriteAt()` must block until the total size goes below that limit.

Multiple implementations are possible:

1. Use `semaphore.Weighted` from `golang.org/x/sync/semaphore`.
2. Implement a throttle controller that has low and high watermarks. It must block `WriteAt()` when the total size of buffers goes above the high watermark, and unblocks it when the total size goes below the low watermark.

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

What a reliable writer needs to do:

1. Collect buffers provided to `WriteAt()` into a scatter-gather buffer.
2. Once there is enough data for a chunk, start a goroutine to write it.
3. If there are too many buffers queued for writing, then throttle the writer.
4. The chunk writer goroutine must retry if a chunk upload fails.

Recall that a chunk upload may be resume from a non-zero offset within a chunk.

Reliable and unreliable writers

ReliableWriter	UnreliableWriter
1. `WriteAt(ctx, buf, off) error` 2. `Complete(ctx) error` 3. `Abort(ctx)`	1. `WriteAt(ctx, chunkBegin, chunkEnd, buf, off) error` 2. `GetResumeOffset(ctx, chunkBegin, chunkEnd) (off, err)` 3. `SetObjectSize(size)` 4. `Complete(ctx) error` 5. `Abort(ctx)`

`UnreliableWriter` is much simpler: `WriteAt()`, `GetResumeOffset()` and `Complete()` just call the corresponding HTTP APIs synchronously.