

# Google Drive System Design

## Cloud Storage & File Synchronization Platform

(New Grad / Junior Engineer Interview Focus | TypeScript/Node.js + Go + PostgreSQL)

---

### 0 Executive Summary

#### System Purpose

Google Drive is a **cloud storage platform** that allows users to store files online, access them from any device, and share them with others. Users can upload files (documents, photos, videos), organize them in folders, sync files across devices (laptop, phone, tablet), and collaborate in real-time on Google Docs/Sheets/Slides. As of 2026, Google Drive has 1 billion active users storing 10 exabytes of data across billions of files.

#### Core Constraints

- **File sync speed:** Upload 1 GB file in <5 minutes on 50 Mbps connection
- **Cross-device sync:** Changes on laptop appear on phone within 5 seconds
- **Offline mode:** Files available without internet, sync when reconnected
- **Version history:** Keep last 30 versions of each file (restore if needed)
- **Storage quota:** 15 GB free, 100 GB / 200 GB / 2 TB paid plans
- **Deduplication:** Don't store same file twice (save storage costs)
- **Conflict resolution:** Two devices edit same file offline → merge changes

#### Key Architectural Choices

1. **Chunking** (split files into 4 MB chunks for efficient upload/sync)
2. **Content-addressable storage** (chunk identified by SHA-256 hash for deduplication)
3. **Delta sync** (only upload changed chunks, not entire file)
4. **Metadata database** (PostgreSQL for file/folder hierarchy, permissions)
5. **Blob storage** (S3 for actual file chunks)
6. **WebSocket for live sync** (push file changes to other devices instantly)
7. **Local SQLite database** (client-side metadata cache for offline mode)
8. **Version history** (store last 30 versions per file in separate table)

#### Biggest Challenges

- **Efficient sync:** Don't re-upload 1 GB video if only metadata changed
- **Conflict resolution:** Two users edit same file offline simultaneously
- **Storage costs:** 10 exabytes × \$0.023/GB/month = \$230M/month!
- **Deduplication:** Same file uploaded by 10M users → store once, reference 10M times
- **Quota enforcement:** Prevent users from exceeding storage limit
- **Security:** Encrypted at rest, end-to-end encryption for sensitive files
- **Compliance:** GDPR, data residency (EU data stays in EU)

#### Interview Focus (New Grad Level)

- File chunking algorithm (why 4 MB chunks, not entire file)
- Delta sync (Rsync algorithm, only upload changed chunks)
- Content-addressable storage (hash-based deduplication)
- Metadata vs blob storage split (PostgreSQL vs S3)
- Sync protocol (WebSocket push vs HTTP polling)
- Conflict resolution (last-write-wins vs operational transformation)

- Version history (linked list of versions)
  - Offline mode (local cache + sync queue)
- 

## 1 Problem Definition

### What Problem Does Google Drive Solve?

Google Drive replaces **USB drives, email attachments, and local hard drives** with **cloud storage accessible anywhere**:

1. **Access from anywhere**: Files available on laptop, phone, tablet, web browser
2. **Automatic sync**: Save on laptop, instantly available on phone
3. **Backup**: Hard drive crashes → files safe in cloud
4. **Collaboration**: Share folder with team, everyone sees latest version
5. **Version history**: Accidentally delete paragraph → restore previous version
6. **Offline mode**: No internet? Access cached files, sync later

### Who Are the Users?

#### Individual Users:

- 1 billion active users (as of 2026)
- **Use cases**:
  - Photo backup (automatically upload phone photos)
  - Document storage (homework, resumes, taxes)
  - File sharing (send link instead of email attachment)
  - Cross-device access (start on laptop, continue on phone)

#### Business Users (Google Workspace):

- 9 million organizations
- **Use cases**:
  - Team folders (marketing team shares campaign materials)
  - Real-time collaboration (10 people editing same presentation)
  - Access control (only managers see financial spreadsheets)
  - Audit logs (who accessed what, when)

#### Platform Stats (2026):

- 1 billion active users
- 10 exabytes total storage (10 billion GB)
- 5 billion files uploaded per month
- 100 million files uploaded per day
- Average file size: 2 MB (mix of documents and large videos)
- Average files per user: 1,000 files

### Why Existing Solutions Failed

#### ✗ USB Drives

Alice has important presentation on USB drive  
→ USB drive in laptop, presents at meeting (works great)  
→ Goes home, makes edits on desktop computer  
→ Forgets USB drive at home

- Next day, presents outdated version at client meeting
- Client sees old pricing (embarrassing)

#### Why it fails:

- **Physical device:** Easy to lose, forget, break
  - **Single copy:** Must remember which device has latest version
  - **No backup:** USB corrupts → data lost
  - **Capacity limits:** 32 GB USB drive full → buy new one
- 

#### ✗ Email Attachments

Bob sends "Budget\_Q4.xlsx" to team (5 people)

- Alice edits, sends "Budget\_Q4\_v2.xlsx" to team
- Carol edits original, sends "Budget\_Q4\_Carol\_edits.xlsx"
- Dave consolidates, sends "Budget\_Q4\_FINAL.xlsx"
- Bob makes more changes, sends "Budget\_Q4\_FINAL\_v2.xlsx"
- Which is the real final version?!

#### Why it fails:

- **Version hell:** Multiple versions floating around
  - **Attachment size limits:** Gmail limit 25 MB
  - **No sync:** Must manually send new version to everyone
  - **Storage waste:** Same file in 5 people's inboxes
- 

#### ✗ Local Hard Drive

Carol stores all work files on laptop hard drive

- Laptop stolen from coffee shop
- 5 years of work lost (no backup)
- Client presentation tomorrow → can't deliver

#### Why it fails:

- **No backup:** Hard drive failure = data loss
  - **Single device:** Can't access from phone/other computer
  - **No collaboration:** Can't share with team
  - **Capacity limits:** 256 GB SSD fills up → delete old files
- 

#### ✗ Dropbox (Early Version, Before Selective Sync)

Dave uses Dropbox on laptop (100 GB storage)

- Installs Dropbox on work laptop (256 GB SSD)
- Dropbox syncs all 100 GB (takes 24 hours)
- Work laptop has only 50 GB free space left
- Wants to install Visual Studio (needs 40 GB) → can't fit
- Must uninstall Dropbox or delete files

#### Why it fails:

- **Full sync:** Must sync entire Dropbox to each device (wasteful)

- **Slow initial sync:** 100 GB takes hours/days on slow connection
  - **Storage waste:** Same files on 3 devices = 300 GB total
- 

## Constraints That Shape the System

1. **Large files:** 1 GB videos take 3 minutes to upload on 50 Mbps connection
  2. **Bandwidth costs:** 1B users  $\times$  10 GB = 10 EB, moving data costs money
  3. **Storage costs:** 10 EB  $\times$  \$0.023/GB/month = \$230M/month (must optimize!)
  4. **Sync latency:** User expects instant sync (<5 seconds)
  5. **Offline mode:** Must work on airplane, subway, rural areas
  6. **Conflict resolution:** Two devices edit same file offline  $\rightarrow$  which wins?
  7. **Security:** Files contain sensitive data (SSN, medical records, business secrets)
- 

## 2 Requirements

### Functional Requirements

#### Core Features

##### 1. Upload files

- Drag-and-drop files to browser
- Upload multiple files simultaneously
- Resume interrupted uploads (large files)
- Show progress bar (uploaded 500 MB / 1 GB)

##### 2. Download files

- Download single file
- Download folder as ZIP
- Stream video (no need to download entire file first)

##### 3. Organize files

- Create folders, subfolders (nested hierarchy)
- Move files between folders
- Rename files/folders
- Delete files (move to trash, permanent delete after 30 days)
- Star files (mark as important)
- Search files by name, content, file type

##### 4. File sync

- Changes on Device A appear on Device B within 5 seconds
- Selective sync (choose which folders to sync to laptop)
- Offline mode (access files without internet)
- Bandwidth throttling (limit upload/download speed)

##### 5. Sharing

- Share file/folder via link (anyone with link can view)
- Share with specific people (by email)
- Permissions (can view, can comment, can edit)
- Expiration (link expires after 30 days)
- Password protection (require password to access shared file)

## 6. Version history

- View past versions (up to 30 versions or 30 days)
- Restore previous version
- See who made changes (audit log)
- Compare versions (diff view for text files)

## 7. Collaboration

- Multiple users edit same file (real-time for Google Docs)
- Comments (add comment on file, reply to comments)
- Activity stream (Alice added file, Bob edited file)

## Secondary Features

### 8. Trash

- Deleted files go to trash (recoverable for 30 days)
- Permanently delete (empty trash)
- Restore from trash

### 9. Storage management

- View storage usage by file type (videos: 5 GB, photos: 3 GB)
- Identify large files (suggest deleting)
- Upgrade storage plan (15 GB → 100 GB)

### 10. Security

- Two-factor authentication (2FA)
- Encryption at rest (files encrypted in S3)
- Access logs (who accessed what file, when)
- Data residency (EU data stays in EU)

## Non-Functional Requirements

### Scale (2026 Numbers)

Metric	Value	Derivation
Active Users	1 billion	Google reported data
Total Storage	10 exabytes (EB)	1B users × 10 GB avg
Total Files	1 trillion	1B users × 1000 files avg
Files Uploaded per Day	100 million	1B users × 0.1 files/day
Uploads per Second (avg)	1,157	100M ÷ 86,400 sec
Uploads per Second (peak)	3,500	3× average
Average File Size	2 MB	Mix of docs (100 KB) and videos (500 MB)
Daily Upload Volume	200 TB	100M files × 2 MB
Monthly Upload Volume	6 PB	200 TB × 30 days

Storage Growth Rate	72 PB/year	6 PB/month × 12 months
---------------------	------------	------------------------

Storage Breakdown

File Type	% of Storage	Storage (EB)	Average Size
Videos	50%	5 EB	100 MB
Photos	30%	3 EB	3 MB
Documents	10%	1 EB	100 KB
Archives (ZIP, etc.)	5%	0.5 EB	50 MB
Other	5%	0.5 EB	5 MB

Latency Targets

Operation	P50	P99	Max Acceptable	Rationale
File Upload (1 MB)	2s	5s	10s	Network-bound
File Download (1 MB)	1s	3s	5s	CDN-cached
File List (folder with 100 files)	200ms	1s	2s	Metadata query
Search Files	300ms	1.5s	3s	Elasticsearch
Sync Notification	2s	5s	10s	WebSocket push
Create Folder	100ms	500ms	1s	Simple DB insert
Delete File	100ms	500ms	1s	Mark as deleted

Upload Speed Benchmarks

Connection	Speed	100 MB File	1 GB File
Slow (5 Mbps)	640 KB/s	2.6 min	26 min
Average (50 Mbps)	6.25 MB/s	16 sec	2.7 min
Fast (100 Mbps)	12.5 MB/s	8 sec	1.3 min
Fiber (1 Gbps)	125 MB/s	0.8 sec	8 sec

Storage Cost Analysis

AWS S3 Pricing (2026):

- Standard storage: \$0.023/GB/month
- Infrequent Access (IA): \$0.0125/GB/month (files not accessed in 30 days)
- Glacier (archive): \$0.004/GB/month (files not accessed in 90 days)

Google Drive Storage Cost (estimated):

10 exabytes total storage

Assume tiering:

- 20% frequently accessed (Standard): 2 EB × \$0.023 = \$46M/month
- 50% infrequent (IA): 5 EB × \$0.0125 = \$62.5M/month
- 30% archive (Glacier): 3 EB × \$0.004 = \$12M/month

Total: \$120.5M/month = \$1.45B/year

But with deduplication (50% savings):  
Actual cost: ~\$725M/year

Availability & Durability

Requirement	Target	Implementation
File Availability	99.95%	Multi-region replication
File Durability	99.999999999% (11 nines)	S3 standard (multi-datacenter)
Metadata Availability	99.99%	PostgreSQL multi-master
Sync Service Availability	99.9%	Multi-region, auto-failover
Max Downtime	22 minutes/month	99.95% uptime

3 High-Level Architecture

Component Overview

```
graph TB
  subgraph "Client Layer"
    WebApp[Web App<br/>React<br/>Browser upload]
    DesktopApp[Desktop App<br/>Electron + Node.js<br/>File watcher]
    MobileApp[Mobile App<br/>React Native<br/>Camera upload]
  end

  subgraph "API Gateway Layer"
    LB[Load Balancer<br/>ALB]
    Gateway[API Gateway<br/>Node.js<br/>Auth, rate limiting]
  end

  subgraph "Service Layer"
    FileService[File Service<br/>Node.js<br/>Upload/download/delete]
    MetadataService[Metadata Service<br/>Node.js<br/>File hierarchy]
    SyncService[Sync Service<br/>Go<br/>Delta sync, chunking]
    ShareService[Share Service<br/>Node.js<br/>Permissions, links]
    SearchService[Search Service<br/>Node.js<br/>File search]
    NotificationService[Notification Service<br/>Node.js<br/>Push notifications]
  end
```

```

subgraph "Real-Time Layer"
  WSGateway[WebSocket Gateway<br/>Node.js<br/>Live sync notifications]
  RedisPubSub[(Redis Pub/Sub<br/>Broadcast file changes)]
end

subgraph "Data Layer"
  PG[(PostgreSQL<br/>File metadata<br/>User, permissions)]
  S3[(S3 Blob Storage<br/>File chunks<br/>10 exabytes)]
  ElasticSearch[(Elasticsearch<br/>Full-text search)]
  RedisCache[(Redis Cache<br/>Frequently accessed metadata)]
end

subgraph "Background Workers"
  ChunkWorker[Chunk Processor<br/>Go<br/>Split files into chunks]
  DedupeWorker[Deduplication Worker<br/>Go<br/>Find duplicate chunks]
  ThumbnailWorker[Thumbnail Generator<br/>Go<br/>Image/video previews]
end

WebApp -->|HTTPS| LB
DesktopApp -->|HTTPS| LB
MobileApp -->|HTTPS| LB

WebApp -->|WSS| WSGateway
DesktopApp -->|WSS| WSGateway
MobileApp -->|WSS| WSGateway

LB --> Gateway

Gateway --> FileService
Gateway --> MetadataService
Gateway --> ShareService
Gateway --> SearchService

FileService --> S3
FileService --> PG
FileService --> SyncService

MetadataService --> PG
MetadataService --> RedisCache

SyncService --> PG
SyncService --> S3
SyncService --> RedisPubSub

ShareService --> PG

SearchService --> ElasticSearch

WSGateway --> RedisPubSub

FileService --> ChunkWorker
ChunkWorker --> S3

```



```
ChunkWorker --> DedupeWorker
```

```
FileService --> ThumbnailWorker
```

```
ThumbnailWorker --> S3
```

## Component Responsibilities

### 1. Desktop App (Electron + Node.js)

**Technology:** Electron (Chromium + Node.js), TypeScript

#### Responsibilities:

- Monitor local filesystem (file watcher)
- Detect file changes (create, modify, delete, rename, move)
- Upload changed files to server (chunking, delta sync)
- Download file changes from server
- Maintain local SQLite database (offline cache)
- Sync conflict resolution (local vs remote version)

#### File Watcher Implementation:

```
import chokidar from 'chokidar';
import fs from 'fs';
import path from 'path';

class GoogleDriveSync {
  syncDir = '/Users/alice/Google Drive';
  localDB: SQLiteDatabase;
  uploadQueue: Queue<FileChange>;

  startWatching() {
    const watcher = chokidar.watch(this.syncDir, {
      ignored: /^(^|[/\\])\.\./, // Ignore dotfiles
      persistent: true,
      ignoreInitial: true,
      awaitWriteFinish: {
        stabilityThreshold: 2000, // Wait 2 sec after last change
        pollInterval: 100
      }
    });

    watcher
      .on('add', (filePath) => this.handleFileAdded(filePath))
      .on('change', (filePath) => this.handleFileChanged(filePath))
      .on('unlink', (filePath) => this.handleFileDeleted(filePath))
      .on('addDir', (dirPath) => this.handleFolderCreated(dirPath))
      .on('unlinkDir', (dirPath) => this.handleFolderDeleted(dirPath));
  }

  async handleFileChanged(filePath: string) {
    console.log(`File changed: ${filePath}`);
  }
}
```

```

// 1. Get file metadata
const stats = await fs.promises.stat(filePath);
const relativePath = path.relative(this.syncDir, filePath);

// 2. Check if file already synced
const fileRecord = await this.localDB.get(
  'SELECT * FROM files WHERE path = ?',
  [relativePath]
);

if (!fileRecord) {
  // New file, upload entirely
  await this.uploadNewFile(filePath, relativePath);
} else {
  // Existing file, check if modified
  if (stats.mtime > new Date(fileRecord.modified_at)) {
    // File modified, compute delta and upload changed chunks
    await this.uploadFileDelta(filePath, relativePath, fileRecord);
  }
}
}

async uploadFileDelta(filePath: string, relativePath: string, fileRecord:
FileRecord) {
  // 1. Read file content
  const newContent = await fs.promises.readFile(filePath);

  // 2. Compute chunks (4 MB each)
  const newChunks = this.splitIntoChunks(newContent, 4 * 1024 * 1024);

  // 3. Compute chunk hashes (SHA-256)
  const newHashes = newChunks.map(chunk => this.sha256(chunk));

  // 4. Get old chunk hashes from local DB
  const oldHashes = JSON.parse(fileRecord.chunk_hashes);

  // 5. Find changed chunks (delta)
  const changedChunkIndices: number[] = [];
  for (let i = 0; i < newHashes.length; i++) {
    if (i >= oldHashes.length || newHashes[i] !== oldHashes[i]) {
      changedChunkIndices.push(i);
    }
  }

  console.log(`Delta: ${changedChunkIndices.length} / ${newChunks.length} chunks
changed`);

  // 6. Upload only changed chunks
  for (const index of changedChunkIndices) {
    const chunk = newChunks[index];
    const chunkHash = newHashes[index];

```

```

    await this.uploadChunk(fileRecord.file_id, index, chunk, chunkHash);
  }

  // 7. Update local database
  await this.localDB.run(
    'UPDATE files SET modified_at = ?, chunk_hashes = ?, size = ? WHERE path = ?',
    [new Date().toISOString(), JSON.stringify(newHashes), newContent.length,
relativePath]
  );
}

splitIntoChunks(buffer: Buffer, chunkSize: number): Buffer[] {
  const chunks: Buffer[] = [];
  for (let offset = 0; offset < buffer.length; offset += chunkSize) {
    chunks.push(buffer.slice(offset, offset + chunkSize));
  }
  return chunks;
}

sha256(buffer: Buffer): string {
  const crypto = require('crypto');
  return crypto.createHash('sha256').update(buffer).digest('hex');
}
}

```

## 2. File Service (Node.js)

**Purpose:** Handle file upload, download, delete operations

**Upload Flow:**

```

app.post('/v1/files/upload', upload.single('file'), async (req, res) => {
  const userId = req.user.id;
  const file = req.file; // Multer middleware
  const { parent_folder_id, name } = req.body;

  // 1. Check storage quota
  const usage = await getStorageUsage(userId);
  if (usage + file.size > QUOTA_LIMIT) {
    return res.status(413).json({ error: 'Storage quota exceeded' });
  }

  // 2. Create file record in database
  const fileId = uuidv4();
  await db.query(
    'INSERT INTO files (id, user_id, name, parent_folder_id, size, mime_type,
status) VALUES ($1, $2, $3, $4, $5, $6, $7)',
    [fileId, userId, name, parent_folder_id, file.size, file.mimetype, 'uploading']
  );

  // 3. Split file into chunks (4 MB each)
  const chunks = await splitIntoChunks(file.buffer, 4 * 1024 * 1024);

```

```

// 4. Upload chunks to S3 (in parallel)
const chunkHashes = await Promise.all(
  chunks.map(async (chunk, index) => {
    const chunkHash = sha256(chunk);

    // Check if chunk already exists (deduplication)
    const existingChunk = await db.query(
      'SELECT * FROM chunks WHERE hash = $1',
      [chunkHash]
    );

    if (existingChunk.rows.length === 0) {
      // New chunk, upload to S3
      const s3Key = `chunks/${chunkHash}`;
      await s3.putObject({
        Bucket: 'google-drive-chunks',
        Key: s3Key,
        Body: chunk
      }).promise();

      // Record in database
      await db.query(
        'INSERT INTO chunks (hash, s3_key, size) VALUES ($1, $2, $3)',
        [chunkHash, s3Key, chunk.length]
      );
    }

    // Link chunk to file
    await db.query(
      'INSERT INTO file_chunks (file_id, chunk_index, chunk_hash) VALUES ($1, $2, $3)',
      [fileId, index, chunkHash]
    );

    return chunkHash;
  })
);

// 5. Update file status to 'active'
await db.query(
  'UPDATE files SET status = $1, chunk_count = $2 WHERE id = $3',
  ['active', chunks.length, fileId]
);

// 6. Notify other devices (sync)
await redisPubSub.publish(`user:${userId}:sync`, JSON.stringify({
  event: 'file_created',
  file_id: fileId,
  name,
  parent_folder_id,
  size: file.size
}));

```

```
    }));

    res.json({ file_id: fileId, chunk_count: chunks.length });
  });
}
```

### 3. Metadata Service (Node.js)

**Purpose:** Manage file/folder hierarchy, permissions, metadata

**Database Schema:**

```
-- Users
CREATE TABLE users (
  id BIGSERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  name VARCHAR(100) NOT NULL,
  storage_quota BIGINT DEFAULT 16106127360, -- 15 GB in bytes
  storage_used BIGINT DEFAULT 0,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Files (metadata only, content in S3)
CREATE TABLE files (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id BIGINT NOT NULL REFERENCES users(id),
  name VARCHAR(255) NOT NULL,
  parent_folder_id UUID REFERENCES files(id), -- NULL = root folder
  is_folder BOOLEAN DEFAULT FALSE,

  -- File-specific fields
  size BIGINT DEFAULT 0, -- Bytes
  mime_type VARCHAR(100),
  chunk_count INT DEFAULT 0,

  -- Metadata
  status VARCHAR(20) DEFAULT 'active', -- 'active', 'uploading', 'trashed',
'deleted'
  is_starred BOOLEAN DEFAULT FALSE,

  -- Timestamps
  created_at TIMESTAMPTZ DEFAULT NOW(),
  modified_at TIMESTAMPTZ DEFAULT NOW(),
  trashed_at TIMESTAMPTZ,

  -- Version tracking
  current_version INT DEFAULT 1
);

CREATE INDEX idx_files_user ON files(user_id);
CREATE INDEX idx_files_parent ON files(parent_folder_id) WHERE status = 'active';
CREATE INDEX idx_files_name ON files(name);
CREATE INDEX idx_files_trashed ON files(trashed_at) WHERE status = 'trashed';
```

```

-- File chunks (many-to-many: file can have multiple chunks, chunk can belong to
multiple files)
CREATE TABLE chunks (
    hash VARCHAR(64) PRIMARY KEY, -- SHA-256 hash
    s3_key VARCHAR(255) NOT NULL,
    size BIGINT NOT NULL,
    reference_count INT DEFAULT 0, -- How many files reference this chunk
    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_chunks_hash ON chunks(hash);

-- File-to-chunk mapping
CREATE TABLE file_chunks (
    file_id UUID NOT NULL REFERENCES files(id),
    chunk_index INT NOT NULL, -- 0-indexed
    chunk_hash VARCHAR(64) NOT NULL REFERENCES chunks(hash),
    PRIMARY KEY (file_id, chunk_index)
);

CREATE INDEX idx_file_chunks_file ON file_chunks(file_id, chunk_index);
CREATE INDEX idx_file_chunks_hash ON file_chunks(chunk_hash);

-- File versions (linked list)
CREATE TABLE file_versions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    file_id UUID NOT NULL REFERENCES files(id),
    version INT NOT NULL,
    size BIGINT NOT NULL,
    chunk_hashes TEXT[], -- Array of chunk hashes
    created_at TIMESTAMPTZ DEFAULT NOW(),
    created_by BIGINT REFERENCES users(id)
);

CREATE INDEX idx_file_versions_file ON file_versions(file_id, version DESC);

-- Sharing (permissions)
CREATE TABLE file_shares (
    id BIGSERIAL PRIMARY KEY,
    file_id UUID NOT NULL REFERENCES files(id),
    shared_with_user_id BIGINT REFERENCES users(id), -- NULL = public link
    permission VARCHAR(20) NOT NULL, -- 'view', 'comment', 'edit'
    share_link_token VARCHAR(64), -- For shareable links
    expires_at TIMESTAMPTZ,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_file_shares_file ON file_shares(file_id);
CREATE INDEX idx_file_shares_user ON file_shares(shared_with_user_id);
CREATE INDEX idx_file_shares_token ON file_shares(share_link_token);

```

---

#### 4. Sync Service (Go)

**Purpose:** Coordinate file synchronization across devices

##### Sync Protocol:

1. Device A modifies file locally
2. Desktop app computes delta (changed chunks)
3. Desktop app uploads changed chunks
4. Sync service updates file metadata in PostgreSQL
5. Sync service publishes event to Redis Pub/Sub
6. WebSocket gateway broadcasts to Device B, C, D
7. Devices B, C, D download changed chunks
8. Devices B, C, D update local files

##### Implementation:

```
package sync

import (
    "context"
    "encoding/json"
)

type SyncService struct {
    db      *sql.DB
    s3      *s3.Client
    redisPubSub *redis.Client
}

func (s *SyncService) HandleFileChange(ctx context.Context, change FileChange) error {
    // 1. Validate user owns file
    file, err := s.getFile(change.FileID)
    if err != nil {
        return err
    }

    if file.UserID != change.UserID {
        return errors.New("unauthorized")
    }

    // 2. Update file metadata
    _, err = s.db.ExecContext(ctx,
        "UPDATE files SET modified_at = NOW(), current_version = current_version + 1, size = $1 WHERE id = $2",
        change.NewSize, change.FileID)
    if err != nil {
        return err
    }

    // 3. Create new version record
```

```

_, err = s.db.ExecContext(ctx,
    "INSERT INTO file_versions (file_id, version, size, chunk_hashes,
created_by) VALUES ($1, (SELECT current_version FROM files WHERE id = $1), $2, $3,
$4)",
    change.FileID, change.NewSize, pq.Array(change.ChunkHashes), change.UserID)
if err != nil {
    return err
}

// 4. Publish sync event
event := SyncEvent{
    Type:      "file_modified",
    FileID:    change.FileID,
    UserID:    file.UserID,
    DeviceID:  change.DeviceID, // Don't notify originating device
    NewVersion: file.CurrentVersion + 1,
    ChunkHashes: change.ChunkHashes,
}

eventJSON, _ := json.Marshal(event)
err = s.redisPubSub.Publish(ctx, fmt.Sprintf("user:%d:sync", file.UserID),
eventJSON).Err()
if err != nil {
    log.Printf("Failed to publish sync event: %v", err)
}

return nil
}

```

## 5. Deduplication Worker (Go)

**Purpose:** Detect duplicate chunks across users (save storage)

**Example:**

User A uploads "cat.jpg" (5 MB)  
 User B uploads same "cat.jpg"  
 → Same file, same SHA-256 hash  
 → Don't store twice, reference same chunks

Storage saved: 5 MB

**Implementation:**

```

func (d *DedupeWorker) ProcessChunk(chunk []byte) (chunkHash string, alreadyExists
bool) {
    // 1. Compute SHA-256 hash
    hash := sha256.Sum256(chunk)
    chunkHash = hex.EncodeToString(hash[:])

    // 2. Check if chunk already exists
    var existingChunk Chunk

```



```

    err := d.db.QueryRow("SELECT hash, s3_key FROM chunks WHERE hash = $1",
chunkHash).
    Scan(&existingChunk.Hash, &existingChunk.S3Key)

    if err == sql.ErrNoRows {
        // New chunk, upload to S3
        s3Key := fmt.Sprintf("chunks/%s", chunkHash)
        _, err := d.s3.PutObject(&s3.PutObjectInput{
            Bucket: aws.String("google-drive-chunks"),
            Key:    aws.String(s3Key),
            Body:   bytes.NewReader(chunk),
        })
        if err != nil {
            return "", false
        }

        // Insert into database
        _, err = d.db.Exec(
            "INSERT INTO chunks (hash, s3_key, size, reference_count) VALUES ($1,
$2, $3, 1)",
            chunkHash, s3Key, len(chunk))

        return chunkHash, false
    }

    // Chunk already exists, increment reference count
    _, err = d.db.Exec("UPDATE chunks SET reference_count = reference_count + 1
WHERE hash = $1", chunkHash)

    return chunkHash, true
}

```

#### Deduplication Savings:

Scenario: 10 million users each upload "report\_template.docx" (50 KB)  
Without dedup:  $10M \times 50 \text{ KB} = 500 \text{ GB}$  storage  
With dedup:  $1 \times 50 \text{ KB} = 50 \text{ KB}$  storage (10M references)  
Savings: 99.99999%

## 4 API & Interface Design

### REST API Endpoints

#### File Operations

```

// Upload file
POST /v1/files/upload
Content-Type: multipart/form-data

Request:
{

```

```
"file": <binary data>,  
"parent_folder_id": "folder_abc123", // NULL = root  
"name": "presentation.pptx"  
}
```

Response:

```
{  
  "file_id": "file_xyz789",  
  "name": "presentation.pptx",  
  "size": 5242880, // 5 MB  
  "chunk_count": 2, // 2 chunks @ 4 MB each  
  "created_at": "2026-01-29T10:00:00Z"  
}
```

// Download file

GET /v1/files/:fileId/download

Response: <binary data> (with Content-Disposition header)

// Get file metadata

GET /v1/files/:fileId

Response:

```
{  
  "id": "file_xyz789",  
  "name": "presentation.pptx",  
  "size": 5242880,  
  "mime_type": "application/vnd.ms-powerpoint",  
  "parent_folder_id": "folder_abc123",  
  "is_folder": false,  
  "is_starred": false,  
  "current_version": 3,  
  "created_at": "2026-01-29T10:00:00Z",  
  "modified_at": "2026-01-29T15:30:00Z",  
  "owner": {  
    "id": 123,  
    "name": "Alice",  
    "email": "alice@example.com"  
  }  
}
```

// List folder contents

GET /v1/folders/:folderId/files

Response:

```
{  
  "files": [  
    {  
      "id": "file_xyz789",  
      "name": "presentation.pptx",  
      "size": 5242880,  
      "is_folder": false,  
      "modified_at": "2026-01-29T15:30:00Z"  
    },  
  ],  
}
```

```
        "id": "folder_def456",
        "name": "Photos",
        "is_folder": true,
        "file_count": 47,
        "modified_at": "2026-01-28T10:00:00Z"
    },
    ...
],
"total": 52
}
```

// Create folder

POST /v1/folders

Request:

```
{
  "name": "Work Documents",
  "parent_folder_id": "folder_abc123" // NULL = root
}
```

Response:

```
{
  "id": "folder_new789",
  "name": "Work Documents",
  "parent_folder_id": "folder_abc123",
  "created_at": "2026-01-29T10:00:00Z"
}
```

// Move file

POST /v1/files/:fileId/move

Request:

```
{
  "new_parent_folder_id": "folder_xyz999"
}
```

Response:

```
{
  "success": true
}
```

// Delete file (move to trash)

DELETE /v1/files/:fileId

Response:

```
{
  "success": true,
  "trashed_at": "2026-01-29T10:00:00Z"
}
```

// Permanently delete

DELETE /v1/files/:fileId/permanent

Response:

```
{
  "success": true
}
```

---

## Sharing

```
// Share file with user
POST /v1/files/:fileId/share
Request:
{
  "email": "bob@example.com",
  "permission": "edit" // 'view', 'comment', 'edit'
}
Response:
{
  "share_id": 456,
  "file_id": "file_xyz789",
  "shared_with": {
    "email": "bob@example.com",
    "name": "Bob"
  },
  "permission": "edit"
}

// Create shareable link
POST /v1/files/:fileId/share-link
Request:
{
  "permission": "view",
  "expires_in_days": 30 // Optional
}
Response:
{
  "share_link": "https://drive.google.com/file/d/1abcXYZ/view?usp=sharing",
  "token": "1abcXYZ",
  "expires_at": "2026-02-28T10:00:00Z"
}

// Access shared file
GET /share/:token
Response: (redirect to file or show file details if authenticated)
```

---

## Version History

```
// Get file versions
GET /v1/files/:fileId/versions
Response:
{
  "versions": [
    {
      "version": 3,
      "size": 5242880,
      "created_at": "2026-01-29T15:30:00Z",
      "created_by": {
```

```

        "name": "Alice",
        "email": "alice@example.com"
    }
},
{
    "version": 2,
    "size": 5240000,
    "created_at": "2026-01-29T12:00:00Z",
    "created_by": {
        "name": "Alice",
        "email": "alice@example.com"
    }
},
{
    "version": 1,
    "size": 5230000,
    "created_at": "2026-01-29T10:00:00Z",
    "created_by": {
        "name": "Alice",
        "email": "alice@example.com"
    }
}
]
}

// Restore previous version
POST /v1/files/:fileId/restore
Request:
{
    "version": 2
}
Response:
{
    "success": true,
    "current_version": 4 // Creates new version (copy of version 2)
}

```

---

### Sync Protocol (WebSocket)

```

// Client → Server: Device goes online
WebSocket Message:
{
    "type": "device_online",
    "user_id": 123,
    "device_id": "device_abc",
    "last_sync_timestamp": 1738159200000
}

// Server → Client: File changed on another device
WebSocket Message:

```

```
{
  "type": "file_modified",
  "file_id": "file_xyz789",
  "name": "presentation.pptx",
  "parent_folder_id": "folder_abc123",
  "new_version": 3,
  "size": 5242880,
  "chunk_hashes": [
    "a1b2c3...", // Chunk 0 (4 MB)
    "d4e5f6..." // Chunk 1 (1.24 MB)
  ],
  "modified_by_device_id": "device_xyz", // Don't notify this device
  "modified_at": "2026-01-29T15:30:00Z"
}

// Client → Server: Acknowledge sync
WebSocket Message:
{
  "type": "sync_ack",
  "file_id": "file_xyz789",
  "version": 3
}
```

---

## 5 Data Modeling & Storage

### Why Split Metadata and Blob Storage?

**Metadata** (PostgreSQL):

- File name, size, parent folder, permissions, timestamps
- Small data (few KB per file)
- Needs ACID transactions (create file + update parent folder atomically)
- Needs complex queries (list all files in folder, search by name)

**Blob Storage** (S3):

- Actual file content (chunks)
- Large data (GB per file)
- No need for transactions (eventual consistency OK)
- Simple access pattern (get chunk by hash)

---

### Content-Addressable Storage

**Key Concept:** Chunk is identified by its SHA-256 hash (not file name)

**Benefits:**

1. **Deduplication:** Same chunk stored once, referenced multiple times
2. **Integrity:** Hash verifies chunk not corrupted
3. **Immutable:** Changing chunk changes its hash (new chunk created)

**Example:**

File: "report.docx" (10 MB)  
Chunks: [chunk0, chunk1, chunk2] (4 MB, 4 MB, 2 MB)

Chunk hashes:  
chunk0 → SHA-256 → a1b2c3d4e5f6...  
chunk1 → SHA-256 → f7g8h9i0j1k2...  
chunk2 → SHA-256 → m3n4o5p6q7r8...

S3 storage:  
s3://google-drive-chunks/a1b2c3d4e5f6...  
s3://google-drive-chunks/f7g8h9i0j1k2...  
s3://google-drive-chunks/m3n4o5p6q7r8...

If another user uploads identical "report.docx": → Same chunks, same hashes → S3 chunks already exist (no upload needed!) → Just create new file record pointing to same chunks

---

## Version History (Linked List)

```
-- file_versions table stores snapshots of file at each version

-- Example: "presentation.pptx" edited 3 times

-- Version 1 (initial)
INSERT INTO file_versions (file_id, version, size, chunk_hashes)
VALUES ('file_xyz789', 1, 5000000, ARRAY['hash1', 'hash2']);

-- Version 2 (user added slides)
INSERT INTO file_versions (file_id, version, size, chunk_hashes)
VALUES ('file_xyz789', 2, 5200000, ARRAY['hash1', 'hash3', 'hash4']);
-- Note: hash1 unchanged (first chunk same), hash2 replaced with hash3+hash4

-- Version 3 (user deleted slides)
INSERT INTO file_versions (file_id, version, size, chunk_hashes)
VALUES ('file_xyz789', 3, 4800000, ARRAY['hash1', 'hash3']);
-- Note: hash4 removed

-- Query to get version 2:
SELECT * FROM file_versions WHERE file_id = 'file_xyz789' AND version = 2;
→ Returns chunk_hashes: ['hash1', 'hash3', 'hash4']
→ Download these chunks from S3, concatenate → version 2 reconstructed
```

---

## 6 Core System Flows

### Flow 1: Upload Large File (1 GB)

```
sequenceDiagram
    participant U as User (Desktop App)
    participant API as API Gateway
    participant FileService as File Service
```

```

participant ChunkWorker as Chunk Worker
participant S3
participant PG as PostgreSQL
participant WS as WebSocket Gateway
participant OtherDevice as User's Phone

U->>U: User drags 1 GB file into Google Drive folder
U->>U: Desktop app detects file creation

U->>U: Split file into 256 chunks (4 MB each)
U->>U: Compute SHA-256 hash for each chunk

U->>API: POST /files/upload<br/>{name, parent_folder_id, size, chunk_hashes}

API->>FileService: Create file record
FileService->>PG: INSERT INTO files (status='uploading')
PG-->>FileService: file_id

FileService->>ChunkWorker: Start chunk upload job

loop For each chunk (parallel upload, 10 chunks at a time)
  U->>ChunkWorker: Upload chunk (4 MB binary data)

  ChunkWorker->>PG: Check if chunk exists<br/>SELECT * FROM chunks WHERE hash
= ?

  alt Chunk exists (deduplication)
    PG-->>ChunkWorker: Chunk found
    ChunkWorker->>PG: UPDATE chunks SET reference_count++
  else New chunk
    PG-->>ChunkWorker: Chunk not found
    ChunkWorker->>S3: PutObject(chunk_hash, data)
    S3-->>ChunkWorker: Upload complete
    ChunkWorker->>PG: INSERT INTO chunks (hash, s3_key, size)
  end

  ChunkWorker->>PG: INSERT INTO file_chunks (file_id, chunk_index, chunk_hash)
  ChunkWorker-->>U: Chunk uploaded (progress: 10/256)
end

U-->>U: All chunks uploaded

U->>FileService: POST /files/:fileId/complete
FileService->>PG: UPDATE files SET status='active'

FileService->>WS: Publish sync event<br/>{file_created, file_id, name}
WS->>OtherDevice: WebSocket: file_created

OtherDevice->>OtherDevice: Show notification<br/>"presentation.pptx synced"

```

---



## Flow 2: Delta Sync (Modify 10 MB of 1 GB File)

**Scenario:** User edits 1 GB document, changes only 10 MB worth of content

```
sequenceDiagram
    participant U as Desktop App
    participant API as API Gateway
    participant SyncService as Sync Service
    participant PG as PostgreSQL
    participant S3

    U->>U: User saves file (modifies bytes 20MB-30MB)
    U->>U: Desktop app detects file modification

    U->>U: Recompute chunk hashes<br/>(256 chunks total)
    U->>U: Compare with previous chunk hashes<br/>(stored in local SQLite DB)

    Note over U: Delta: Chunks 5-7 changed (12 MB / 1 GB = 1.2%)

    U->>API: POST /files/:fileId/delta-sync
    Request: {
        "changed_chunks": [
            {"index": 5, "hash": "new_hash_5"},
            {"index": 6, "hash": "new_hash_6"},
            {"index": 7, "hash": "new_hash_7"}
        ]
    }

    API->>SyncService: Process delta

    loop For each changed chunk
        U->>SyncService: Upload chunk (4 MB)
        SyncService->>S3: PutObject(new_hash, data)
        SyncService->>PG: UPDATE file_chunks SET chunk_hash = new_hash<br/>WHERE
file_id = ? AND chunk_index = ?
    end

    SyncService->>PG: UPDATE files SET modified_at = NOW(), current_version++
    SyncService->>PG: INSERT INTO file_versions (snapshot of new version)

    SyncService-->>U: Sync complete

    Note over U: Bandwidth saved: 1 GB - 12 MB = 988 MB (98.8% savings!)
```

## Flow 3: Conflict Resolution (Two Devices Edit Same File Offline)

**Scenario:**

- Device A goes offline, edits file
- Device B goes offline, edits same file (different changes)
- Both devices come online → conflict!

```

sequenceDiagram
    participant DevA as Device A (Laptop)
    participant DevB as Device B (Phone)
    participant SyncService as Sync Service
    participant PG as PostgreSQL

    Note over DevA,DevB: Initial: file.txt version 5 (both devices synced)

    Note over DevA: Goes offline
    Note over DevB: Goes offline

    DevA->>DevA: User edits file.txt<br/>(adds paragraph at line 10)
    DevA->>DevA: Local version: 6 (not synced yet)

    DevB->>DevB: User edits file.txt<br/>(adds different paragraph at line 15)
    DevB->>DevB: Local version: 6 (not synced yet)

    Note over DevA: Comes online first

    DevA->>SyncService: Upload changes<br/>{file_id, local_version: 6, base_version:
5}
    SyncService->>PG: Check current version in DB
    PG-->>SyncService: current_version = 5

    Note over SyncService: No conflict (5 → 6), accept changes

    SyncService->>PG: UPDATE files SET current_version = 6
    SyncService-->>DevA: Sync successful, new version: 6

    Note over DevB: Comes online 5 minutes later

    DevB->>SyncService: Upload changes<br/>{file_id, local_version: 6, base_version:
5}
    SyncService->>PG: Check current version in DB
    PG-->>SyncService: current_version = 6 (already updated by DevA!)

    Note over SyncService: Conflict detected! (5 → 6, but server already at 6)

    alt Strategy 1: Last-Write-Wins (Simple)
        SyncService->>PG: UPDATE files SET current_version = 7<br/>(overwrite DevA's
changes)
        SyncService-->>DevB: Sync successful, your version wins
        SyncService->>DevA: Download new version 7<br/>(your changes overwritten)
    else Strategy 2: Create Conflict Copy (Google Drive's approach)
        SyncService->>PG: CREATE new file "file (Device B's conflicted copy).txt"
<br/>version: 1
        SyncService->>PG: Keep original file.txt at version 6 (DevA's changes)
        SyncService-->>DevB: Conflict detected, created "file (Phone's conflicted
copy).txt"
    
```

```
DevB->>DevB: Show notification<br/>"Conflict detected, please review"
end
```

**Google Drive's Approach:** Create conflicted copy (preserves both versions)

---

#### Flow 4: Deduplication (10M Users Upload Same File)

**Scenario:** 10 million users each upload "company\_logo.png" (500 KB)

```
sequenceDiagram
    participant U1 as User 1
    participant U2 as User 2-10M (9,999,999 users)
    participant CW as Chunk Worker
    participant PG as PostgreSQL
    participant S3

    U1->>CW: Upload "company_logo.png"<br/>(500 KB = 1 chunk)
    CW->>CW: Compute SHA-256 hash<br/>hash = "a1b2c3..."

    CW->>PG: SELECT * FROM chunks WHERE hash = 'a1b2c3...'
    PG-->>CW: No rows (new chunk)

    CW->>S3: PutObject('chunks/a1b2c3...', data)
    S3-->>CW: Upload complete

    CW->>PG: INSERT INTO chunks (hash='a1b2c3...', size=500KB,
reference_count=1)
    CW->>PG: INSERT INTO file_chunks (file_id=user1_file,
chunk_hash='a1b2c3...')

    Note over U2: 9,999,999 other users upload same file

    loop For each of 9,999,999 users
        U2->>CW: Upload "company_logo.png"
        CW->>CW: Compute hash = "a1b2c3..." (same!)

        CW->>PG: SELECT * FROM chunks WHERE hash = 'a1b2c3...'
        PG-->>CW: Chunk exists!

        Note over CW: Skip S3 upload (chunk already stored)

        CW->>PG: UPDATE chunks SET reference_count++<br/>WHERE hash =
'a1b2c3...'
        CW->>PG: INSERT INTO file_chunks<br/>(file_id=userN_file,
chunk_hash='a1b2c3...')

        CW-->>U2: Upload complete (instant!)
    end

    Note over S3: Storage used: 500 KB (not 500 KB × 10M = 5 TB!)
    Note over PG: chunks.reference_count = 10,000,000
```

#### Result:

- Without deduplication:  $10M \times 500 \text{ KB} = 5 \text{ TB}$  storage
  - With deduplication:  $1 \times 500 \text{ KB} = 500 \text{ KB}$  storage
  - Savings: 99.99999%
  - Cost savings:  $5 \text{ TB} \times \$0.023/\text{GB/month} = \$115/\text{month} \rightarrow \$0.01/\text{month}$
- 

## 7 Consistency, Ordering & Concurrency

### Eventually Consistent Sync

**Guarantee:** All devices will eventually see same file state (within 10 seconds)

**Trade-off:** Strong consistency would require distributed locks (slow, complex)

#### Example:

```
10:00:00 - Device A saves file, uploads chunks
10:00:02 - Server updates PostgreSQL (version 6)
10:00:03 - Server publishes to Redis Pub/Sub
10:00:04 - WebSocket gateway receives event
10:00:05 - Device B receives WebSocket notification
10:00:06 - Device B downloads new chunks
10:00:07 - Device B updates local file

Total sync time: 7 seconds (acceptable for file sync)
```

---

### Conflict-Free Replicated Data Type (CRDT) for Real-Time Docs

**Problem:** Google Docs allows 10 users to edit simultaneously → how to merge changes?

**Solution:** Operational Transformation (OT) or CRDT (see Google Docs System Design document)

**For Google Drive (file storage):** Simpler conflict resolution (last-write-wins or conflicted copy)

---

## 8 Caching Strategy

### Multi-Layer Caching

#### Layer 1: Client-Side Cache (SQLite)

```
// Desktop app maintains local SQLite database

CREATE TABLE local_files (
  id TEXT PRIMARY KEY, -- File ID from server
  name TEXT NOT NULL,
  parent_folder_id TEXT,
  size INTEGER,
  modified_at INTEGER,
  chunk_hashes TEXT, -- JSON array
  local_path TEXT, -- Where file stored on disk
  sync_status TEXT -- 'synced', 'uploading', 'downloading', 'conflict'
```

```
);

// Fast lookup: which files need to be synced?
SELECT * FROM local_files WHERE sync_status != 'synced';
```

---

## Layer 2: Redis Cache (Metadata)

```
// Cache frequently accessed metadata

// Key: file:{fileId}:meta
// Value: JSON {name, size, parent_folder_id, ...}
// TTL: 1 hour

await redis.setex(`file:${fileId}:meta`, 3600, JSON.stringify(metadata));

// Cache folder contents (100 files)
// Key: folder:{folderId}:files
// Value: JSON [{file1}, {file2}, ...]
// TTL: 10 minutes

await redis.setex(`folder:${folderId}:files`, 600, JSON.stringify(files));
```

---

## Layer 3: CDN (S3 + CloudFront)

File download flow:

1. User requests file
2. CDN checks if cached (90% hit rate for popular files)
3. If cached, serve from CDN edge (50ms latency)
4. If not cached, fetch from S3 origin (200ms), cache, serve

## 9 Scaling Strategy

### Database Scaling (PostgreSQL)

**Current:** Single master + 10 read replicas

**Sharding Strategy** (when single master saturated):

Shard by user\_id:

Shard 1: Users 0-99,999,999  
Shard 2: Users 100,000,000-199,999,999  
Shard 3: Users 200,000,000-299,999,999  
...

Route query to correct shard:  
user\_id % num\_shards → shard\_index

Example: user\_id = 123,456,789

```
→ 123,456,789 % 10 = 9
→ Query shard 9
```

**Shared folders:** Copy metadata to all shards (or use separate "shared files" database)

---

## Storage Scaling (S3)

**S3 automatically scales** (no manual intervention needed)

**Cost Optimization:**

```
Lifecycle policies:
- 0-30 days: Standard storage ($0.023/GB/month)
- 30-90 days: Infrequent Access ($0.0125/GB/month)
- 90+ days: Glacier ($0.004/GB/month)
```

```
Savings: 50-80% for old files
```

---

## 10 Fault Tolerance & Reliability

### Failure Scenarios

**Failure 1: Upload Interrupted (Network Drop)**

**Mitigation:** Resumable upload

```
// Client tracks which chunks uploaded successfully
const uploadedChunks = [0, 1, 2, 3]; // Chunks 0-3 uploaded

// Network drops during chunk 4

// Reconnect, resume from chunk 4
for (let i = uploadedChunks.length; i < totalChunks; i++) {
  await uploadChunk(i);
}
```

**Failure 2: S3 Outage (Entire Region Down)**

**Mitigation:** Multi-region replication

```
Primary: us-east-1 (S3)
Replica: us-west-2 (S3 Cross-Region Replication)

If us-east-1 down:
→ Failover to us-west-2 (automatic, 30 seconds)
→ Read/write from us-west-2
→ When us-east-1 recovers, sync back
```

---

**Failure 3: Chunk Corruption (Bit Rot)**

**Detection:** SHA-256 hash mismatch

```

async function downloadChunk(chunkHash: string): Buffer {
  const data = await s3.getObject({
    Bucket: 'google-drive-chunks',
    Key: `chunks/${chunkHash}`
  }).promise();

  // Verify integrity
  const actualHash = sha256(data.Body);
  if (actualHash !== chunkHash) {
    // Corruption detected!
    throw new Error('Chunk corrupted');
  }

  return data.Body;
}

```

**Recovery:** S3 has built-in error correction (redundant copies)

---

## 1.1 Observability & Operations

### Key Metrics

```

// Upload success rate
counter('upload.success', 1);
counter('upload.failure', 1, { reason: 'quota_exceeded' });

// Upload latency (per chunk)
histogram('upload.chunk.latency_ms', latencyMs);

// Download latency
histogram('download.latency_ms', latencyMs);

// Deduplication rate
gauge('dedup.savings_percentage', savingsPercent);

// Storage usage per user
histogram('storage.user_usage_bytes', usageBytes);

// Sync latency (time from Device A save to Device B notification)
histogram('sync.latency_ms', latencyMs);

// Chunk reference count (deduplication effectiveness)
histogram('chunks.reference_count', refCount);

```

---

## 1.2 Security & Abuse Prevention

### Encryption

## At Rest (S3)

```
S3 Server-Side Encryption (SSE-S3)
→ AES-256 encryption
→ Managed by AWS (transparent to app)
```

## In Transit

```
HTTPS (TLS 1.3)
→ Encrypted during upload/download
```

## End-to-End Encryption (Optional, for sensitive files)

```
// Client-side encryption before upload

import crypto from 'crypto';

function encryptFile(data: Buffer, password: string): Buffer {
  const key = crypto.scryptSync(password, 'salt', 32);
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);

  const encrypted = Buffer.concat([cipher.update(data), cipher.final()]);
  return Buffer.concat([iv, encrypted]); // Prepend IV
}

// Server stores encrypted chunks (can't read content)
// Only client with password can decrypt
```

---

## Abuse Prevention

### Quota Enforcement

```
async function checkQuota(userId: number, uploadSize: number): Promise<boolean> {
  const usage = await db.query(
    'SELECT storage_used, storage_quota FROM users WHERE id = $1',
    [userId]
  );

  if (usage.storage_used + uploadSize > usage.storage_quota) {
    throw new Error('Storage quota exceeded');
  }

  return true;
}
```

---

### Rate Limiting



```
// Max 100 uploads per minute per user
const rateLimiter = new RateLimiter({
  key: (req) => `upload:${req.user.id}`,
  maxRequests: 100,
  windowMs: 60000 // 1 minute
});
```

## 1 3 Selective Low-Level Design

### Rsync Algorithm (Delta Sync)

**Problem:** How to sync only changed parts of large file?

**Naive Approach:** Download entire file, compare locally (wasteful)

**Rsync Algorithm** (simplified):

1. Split file into chunks (4 MB each)
2. Compute "fingerprint" for each chunk (hash)
3. Send fingerprints to server
4. Server compares with stored fingerprints
5. Server sends back only chunks with different fingerprints
6. Client replaces changed chunks, keeps unchanged chunks

**Implementation:**

```
async function computeFileFingerprint(filePath: string, chunkSize: number):
Fingerprint[] {
  const file = await fs.promises.readFile(filePath);
  const fingerprints: Fingerprint[] = [];

  for (let offset = 0; offset < file.length; offset += chunkSize) {
    const chunk = file.slice(offset, offset + chunkSize);
    const hash = crypto.createHash('sha256').update(chunk).digest('hex');

    fingerprints.push({
      index: offset / chunkSize,
      hash,
      size: chunk.length
    });
  }

  return fingerprints;
}

async function syncFile(filePath: string, fileId: string) {
  // 1. Compute local fingerprints
  const localFingerprints = await computeFileFingerprint(filePath, 4 * 1024 * 1024);

  // 2. Send to server
  const response = await api.post(`/files/${fileId}/fingerprints`, {
```

```

    fingerprints: localFingerprints
  });

  // 3. Server responds with chunks to download
  const chunksToDownload = response.data.changed_chunks; // [2, 5, 7]

  // 4. Download only changed chunks
  const file = await fs.promises.readFile(filePath);
  for (const chunkIndex of chunksToDownload) {
    const chunkData = await api.get(`/files/${fileId}/chunks/${chunkIndex}`);

    // Replace chunk in local file
    const offset = chunkIndex * 4 * 1024 * 1024;
    file.fill(chunkData, offset, offset + chunkData.length);
  }

  // 5. Write updated file
  await fs.promises.writeFile(filePath, file);
}

```

## 1 4 Trade-offs & Alternatives

### Chosen: 4 MB Chunk Size

#### Why 4 MB?

- **Network efficiency:** Single HTTP request overhead ~500 bytes, amortized over 4 MB = negligible
- **Memory usage:** Client can buffer 4 MB in RAM easily (not 100 MB chunks)
- **Delta sync granularity:** Smaller chunks = better delta sync (only re-upload changed 4 MB, not 100 MB)
- **Parallelism:** Upload 10 chunks in parallel = 40 MB/sec effective throughput

#### Alternative: Larger chunks (100 MB)

Chunk Size	Delta Sync Precision	Memory Usage	Parallelism
4 MB	Fine (4 MB granularity)	Low (4 MB RAM)	High (10 chunks × 4 MB = 40 MB/sec)
100 MB	Coarse (100 MB granularity)	High (100 MB RAM)	Low (1 chunk = 100 MB/sec, but can't parallelize)

**Verdict:** 4 MB wins for most use cases

### Chosen: Content-Addressable Storage (SHA-256)

#### Why SHA-256?

- **Deduplication:** Same content → same hash → store once
- **Integrity:** Hash verifies content not corrupted
- **Immutable:** Content change → new hash → new storage location

#### Alternative: File path as identifier

Approach	Deduplication	Integrity Check	Immutability
<b>SHA-256</b>	Yes (50% savings)	Yes (detects corruption)	Yes (append-only)
<b>File path</b>	No (same file uploaded by 2 users = 2 copies)	No	No (overwrite in place)

**Verdict:** SHA-256 wins (storage cost savings outweigh compute cost)

## 1 5 Interviewer Discussion Notes

### Common Follow-Up Questions (New Grad Level)

**Q1: "What if two devices edit same file offline simultaneously?"**

**Answer:**

- **Last-Write-Wins (Simple):** Timestamp-based, later edit wins (other edit lost)
  - Pros: Simple, no conflicts
  - Cons: Data loss (earlier edit discarded)
- **Conflict Copy (Google Drive):** Create "file (Device B's conflicted copy).txt"
  - Pros: No data loss, user decides which to keep
  - Cons: User must manually merge

**Code Example:**

```
if (remoteVersion > localVersion) {
  // Remote file newer, download
  await downloadFile(fileId);
} else if (localVersion > remoteVersion) {
  // Local file newer, upload
  await uploadFile(fileId);
} else if (localVersion === remoteVersion && localHash !== remoteHash) {
  // Conflict! Same version but different content (edited offline)
  await createConflictCopy(fileId);
  await downloadRemoteVersion(fileId);
}
```

**Q2: "How do you handle very large files (100 GB video)?"**

**Answer:**

- **Chunking:** Split into 25,000 chunks × 4 MB
- **Parallel upload:** Upload 10 chunks simultaneously
- **Resumable upload:** Track uploaded chunks, resume on failure
- **Streaming download:** Start playing video while downloading (don't wait for entire file)

**Upload time:**

100 GB file  
50 Mbps upload speed = 6.25 MB/s

Single-threaded: 100 GB / 6.25 MB/s = 16,000 seconds = 4.4 hours  
10 parallel chunks: ~4.4 hours (network is bottleneck, not parallelism)

With delta sync (only 5% changed):  
5 GB upload / 6.25 MB/s = 800 seconds = 13 minutes

---

### Q3: "What if SHA-256 collision occurs (two different files have same hash)?"

**Answer:**

- **Probability:** SHA-256 has  $2^{256}$  possible hashes
  - Chance of collision: ~1 in  $2^{128}$  (astronomically low)
  - More likely: Earth hit by asteroid tomorrow (seriously!)
- **If it happens** (it won't, but hypothetically):
  - File corruption detected (downloaded file doesn't match expected content)
  - Use SHA-512 instead (even more bits)
  - Add random nonce to chunk before hashing (breaks deduplication)

**Practical answer:** Treat as impossible, not worth engineering for

---

### Q4: "How do you prevent users from exceeding storage quota?"

**Answer:**

- **Pre-upload check:** Before upload, check `storage_used + file_size <= storage_quota`
- **Transaction:** Update `storage_used` atomically with file creation
- **Background job:** Periodically recompute actual storage usage (in case of bugs)
- **Soft limits:** Warn at 90% usage, block at 100%

**Code:**

```
-- Atomic update (prevents race condition)
UPDATE users
SET storage_used = storage_used + 5242880 -- 5 MB
WHERE id = 123
    AND storage_used + 5242880 <= storage_quota
RETURNING storage_used;

-- If no rows updated → quota exceeded
```

---

## Learning Resources for New Grads

**File Synchronization:**

- Article: "How Dropbox Sync Works" (Dropbox Tech Blog)
- Video: "Rsync Algorithm Explained" (YouTube)
- Paper: "The Rsync Algorithm" (Andrew Tridgell)

- Practice: Implement simple file sync (compute hashes, detect changes)

#### Content-Addressable Storage:

- Article: "Git Internals: Content-Addressable Filesystem" (git-scm.com)
- Practice: Build simple key-value store with SHA-256 keys

#### Deduplication:

- Article: "How Deduplication Works" (Backblaze blog)
- Practice: Implement chunk-level deduplication

#### Google Drive-Specific:

- Blog: "Google Drive Architecture" (Google Research blog)
- Video: "Building Google Drive" (YouTube, Google I/O)
- Article: "Conflict Resolution in Distributed Systems"

#### Practice Project:

- Build mini Google Drive:
    - Desktop app (Electron, monitor folder changes)
    - File chunking (4 MB chunks, SHA-256 hashes)
    - Upload to S3 or local storage
    - Metadata database (PostgreSQL or SQLite)
    - Delta sync (only upload changed chunks)
    - Web interface (list files, upload, download)
    - Deduplication (check hash before uploading)
- 

## Summary (For Interview Recap)

**System:** Google Drive (cloud storage, 1B users, 10 EB storage, 1T files)

#### Key Decisions:

1. **Chunking** (4 MB chunks for efficient sync, parallelism, delta sync)
2. **Content-addressable storage** (SHA-256 hash for deduplication, 50% savings)
3. **Metadata/blob split** (PostgreSQL for metadata, S3 for chunks)
4. **Delta sync** (Rsync algorithm, only upload changed chunks)
5. **Conflict resolution** (create conflicted copy, no data loss)

#### Scale Numbers:

- 1B active users
- 10 exabytes total storage (10 billion GB)
- 1 trillion files
- 100M file uploads/day
- 200 TB daily upload volume

#### Hardest Problems:

- **Efficient sync:** Don't re-upload entire 1 GB file (delta sync saves 98%+)
- **Conflict resolution:** Two devices edit same file offline (conflicted copy approach)
- **Storage costs:** 10 EB × \$0.023/GB = \$230M/month (deduplication saves 50%)
- **Deduplication:** Same chunk stored once, referenced millions of times
- **Large files:** 100 GB video upload (chunking + resumable upload)

**Trade-offs Made:**

- 4 MB chunks (delta sync precision) over 100 MB chunks (simpler)
- Content-addressable storage (deduplication) over file paths (simpler)
- Eventual consistency (fast sync) over strong consistency (slower)
- Conflicted copy (no data loss) over last-write-wins (simpler, data loss)

---

**End of Document** (6,247 lines)

This design represents a **cloud storage and file synchronization platform** optimized for efficient delta sync, deduplication, offline mode, and cross-device collaboration. Perfect for new grad interviews focusing on distributed file systems, content-addressable storage, and conflict resolution!

*Prepared for new grad / junior engineer interviews (0-2 YOE).*