HLD Case Study 6 (Design Hotstar)

Contents

[Hotstar Design Overview 2](#_Toc184928840)

[Understanding Images and Videos 5](#_Toc184928841)

[Hotstar's Data Storage Design 5](#_Toc184928842)

[Database Design Considerations 6](#_Toc184928843)

[Estimation of Scale 6](#_Toc184928844)

[Hotstar's Data Storage Design 8](#_Toc184928845)

[Estimation and Storage for Movies Metadata 8](#_Toc184928846)

[Estimation and Storage for User Data 8](#_Toc184928847)

[Use Cases for Hotstar-like Video Streaming Service 10](#_Toc184928848)

[Playback Implementation Details 12](#_Toc184928849)

[Video Streaming 13](#_Toc184928850)

[Some Points 17](#_Toc184928851)

[Architecture for Live Streaming 19](#_Toc184928852)

[Real-Time Video Architecture and Related Concepts 22](#_Toc184928853)

[Doubt Clearing Session Questions 24](#_Toc184928854)

## Hotstar Design Overview

The lecture discusses designing an OTT platform like Hotstar, focusing on functional and non-functional goals, estimation of scale, API design, and final system architecture. The emphasis is on understanding core user functionalities and exploring technical challenges related to video streaming and live events.

**Key Discussion Points**

1. **Steps in System Design**:
   * **MVP Definition**:
     + Define essential functionalities (e.g., user interface for watching and searching for content).
   * **Scale Estimation**:
     + Understand the data scale, concurrent user handling, and need for sharding.
   * **API Design**:
     + Define APIs for backend interactions.
   * **Final Design**:
     + Detailed system architecture covering storage, data flow, and optimizations.

Today's focus is on the final design, particularly from the user's perspective.

**Segmentation of Hotstar Functionality**

1. **Content Upload**:
   * Interface for content providers to upload movies, TV shows, etc.
   * Metadata management:
     + Video details (thumbnails, subtitles, audio tracks).
     + Rights and resolution information.
     + Language options.
2. **User Experience**:
   * Discover content via search or browsing.
   * Play video with adaptive quality based on internet speed.
   * Ensure minimal buffering and smooth playback.
   * Resume content from the last watched point.
3. **Live Streaming**:
   * Stream live events (e.g., sports matches) with minimal lag.
   * Enable replay or rewind even during live events.
   * Features like preview thumbnails when scrolling through a stream.

**Focus Areas for User Experience Design**

1. **Content Discovery**:
   * Efficient cataloguing of movies and TV shows.
   * Exclude advanced features like recommendation systems for MVP.
2. **Video Playback**:
   * Adaptive streaming for different resolutions.
   * Techniques to reduce buffering:
     + Efficient caching and pre-loading strategies.
   * Data optimization for mobile devices.
3. **Subtitles and Audio Tracks**:
   * Support multiple languages for audio and subtitles.
   * Ensure synchronization during playback.
4. **Advanced Features**:
   * Resume playback functionality.
   * Preview frames while scrubbing through the timeline.

**Technical Deep Dive**

1. **Image Representation**:
   * **Black-and-white Images**:
     + Represented as a matrix of 0s and 1s (pixels).
     + Higher resolution achieved by increasing the number of pixels.
   * **Compression**:
     + Formats like JPEG/PNG compress image data.
     + Techniques include representing contiguous regions with simplified data (e.g., Huffman encoding).
2. **Video Representation**:
   * **Videos as Frames**:
     + A video is a series of images (frames) played sequentially.
     + Frame rate controls video speed.
   * **Delta Encoding**:
     + Instead of storing each frame, store differences (deltas) between consecutive frames.
     + Reduces storage space while maintaining playback continuity.
3. **Adaptive Streaming**:
   * Dynamically adjust video quality based on network conditions.
   * Ensure lightweight streaming on client devices.

**Live Streaming Challenges**

1. **Latency**:
   * Minimize lag between live event and user view.
2. **Rewind and Replay**:
   * Support going back during live streams.
   * Store and serve segments dynamically as the event progresses.
3. **Subtitles and Audio Sync**:
   * Maintain synchronization in real-time.

## Understanding Images and Videos

**Basics of Image Representation**

* **Images as Matrices:**
  + Images and videos are represented as matrices, where each pixel corresponds to a matrix element.
  + For black-and-white images, pixel values are binary (0 or 1).
  + For coloured images, each pixel has RGB (Red, Green, Blue) values ranging from 0 to 255.
  + Higher pixel density results in better granularity and resolution.
* **Videos as Collections of Images:**
  + A video is a sequence of images (frames) over time.
  + Files like images and videos are encoded and compressed for storage.

**Types of Files Associated with Videos**

1. **Video Files:** Actual movie or TV show files.
2. **Thumbnails:** Multiple thumbnails may be required, e.g., for timestamps.
3. **Trailers/Teasers:** Promo files that autoplay for previews.
4. **Subtitles:** Text files for dialogue in different languages.
5. **Audio Files:** For multiple language tracks or audio descriptions.

## Hotstar's Data Storage Design

**Categories of Data**

1. **User Data:**
   * Includes user profiles, subscription information, watch history, and preferences (e.g., watchlists).
2. **Metadata for Movies/TV Shows:**
   * Title, thumbnail URL, genres, duration, actors, languages, and other attributes.
   * Example databases: Tables for movies, actors, movie-actors, movie-languages.
3. **Large Files:**
   * Includes video files, thumbnails, trailers, audio, and subtitles.
   * Stored on specialized file systems (e.g., S3, HDFS) and cached in a CDN for fast access.

## Database Design Considerations

**Separation of Concerns**

* Different data types stored in separate databases:
  + Metadata: Stored in relational databases like MySQL.
  + Large Files: Stored in distributed file systems.
  + User Information: Sharded relational databases or NoSQL solutions.

**Sharding in Databases**

* **Sharding for Scalability:**
  + User database: Likely to need sharding due to large volume.
  + Movie database: May fit in a single machine, depending on scale.
* **Manual Sharding:**
  + Sharding in MySQL requires custom logic, as it does not support sharding out-of-the-box.
  + Example: Consistent hashing for user ID-based sharding.

## Estimation of Scale

**Movies and TV Shows**

* **Movies:**
  + Example estimation based on Hollywood: ~200 movies/year over 50 years = ~10,000 movies.
  + Scaling globally with ~10 industries = ~280,000 movies.
* **TV Shows/Web Series:**
  + Assume similar scale as movies: ~280,000 TV shows.
* **Combined Content:**
  + Total estimate: ~840,000 items (movies, TV shows, web series).
  + For Hotstar's 20% market share: ~170,000 items.

**Content Metadata Storage**

* Metadata (title, genre, actors, etc.) fits in a relational database.
* Searching within metadata can be handled by MySQL or specialized search tools like Elasticsearch for full-text queries.

**Challenges and Tools**

* **MapReduce:**
  + Used for analytics and asynchronous jobs, not for live queries.
  + MapReduce jobs are slow and resource-intensive.
* **Hive and HBase:**
  + Hive queries transform into MapReduce jobs, making them unsuitable for live queries.
* **ElasticSearch:**
  + Suitable for advanced searches, e.g., querying actors, scenes, or descriptions.

# Hotstar's Data Storage Design

## Estimation and Storage for Movies Metadata

* **Key Metadata Parameters for Each Movie:**
  1. Duration.
  2. Actors involved (IDs stored).
  3. Supported languages.
  4. URLs for subtitle and audio files (stored as addresses, not actual files).
  5. Description (e.g., 500-word synopsis).
  6. IMDb rating (optional).
  7. Maturity rating (e.g., adult or non-adult content).
* **Storage Requirements Calculation:**
  1. **Parameters:** 20 parameters at 8 bytes each – **160 bytes**.
  2. **Description:** 500 bytes.
  3. **Actors:** Up to 50 actors (IDs as 8-byte integers) – **400 bytes**.
  4. **Subtitle and Audio URLs:** Approximately 500 bytes.
  5. **Total per Movie:** 160 + 500 + 400 + 500 = **2 KB**.
  6. **Total for 200,000 Movies:** 200,000 \* 2 KB = **400 MB**.
* **Conclusion:**
  1. The metadata for 200,000 movies (~400 MB) can easily fit into a single MySQL instance.
  2. No sharding is required for storing movie metadata.

## Estimation and Storage for User Data

* **Key Attributes for Each User:**
  1. **Watchlist:** List of movie IDs (on average, 2-3 movie IDs per user).
  2. **Watched Movies:** List of started/completed movies with timestamps (on average, 50 movies per user).
  3. **User Attributes:** Email ID, phone number, subscription plan, end date, etc.
* **Storage Requirements Calculation:**
  1. **Watchlist:** 2-3 movie IDs per user (8 bytes each) – **24 bytes**.
  2. **Watched Movies:**
     + 50 movie IDs (8 bytes each).
     + Timestamp for each movie (8 bytes).
     + Total: 50 \* (8 + 8) = **800 bytes**.
  3. **User Attributes:** Approximately **1 KB**.
  4. **Total per User:** 24 + 800 + 1,000 = ~**2 KB**.
  5. **Total for 60 Million Users:** 60,000,000 \* 2 KB = **120 GB**.
* **Conclusion:**
  1. User data (~120 GB) can also fit into a single MySQL instance.
  2. No sharding is required for storing user data. Replicas and caching can be used for scalability.

**Read vs. Write Analysis**

* **Read Operations:**
  + Searching for a movie.
  + Loading metadata for movies and user watchlists.
  + Streaming-related reads (e.g., fetching metadata for video chunks).
* **Write Operations:**
  + Adding a movie to the watchlist.
  + Recording the start or progress of watching a movie.
  + Uploading new content (infrequent operation).
* **System Characteristics:**
  + **Read-heavy system.**
  + Most operations (e.g., querying, streaming) are reads.
  + Writes are less frequent and primarily occur during user updates or new content uploads.
* **Conclusion:**
  + MySQL with appropriate replicas and caching mechanisms works well for this read-heavy workload.

**Key Takeaways:**

1. **Movie Metadata:**
   * Total size: 400 MB.
   * Easily stored on a single MySQL instance without sharding.
2. **User Data:**
   * Total size: 120 GB.
   * Can also fit on a single MySQL instance with some replicas.
3. **Sharding:**
   * Not required for either movie metadata or user data at this scale.
4. **System Design Consideration:**
   * As a read-heavy system, caching and replication will optimize performance effectively.
5. **Scale Assumptions:**
   * 200,000 movies.
   * 60 million users with average activity assumptions (watchlists, watched movies).

## Use Cases for Hotstar-like Video Streaming Service

**Primary Use Cases (Focus Areas)**

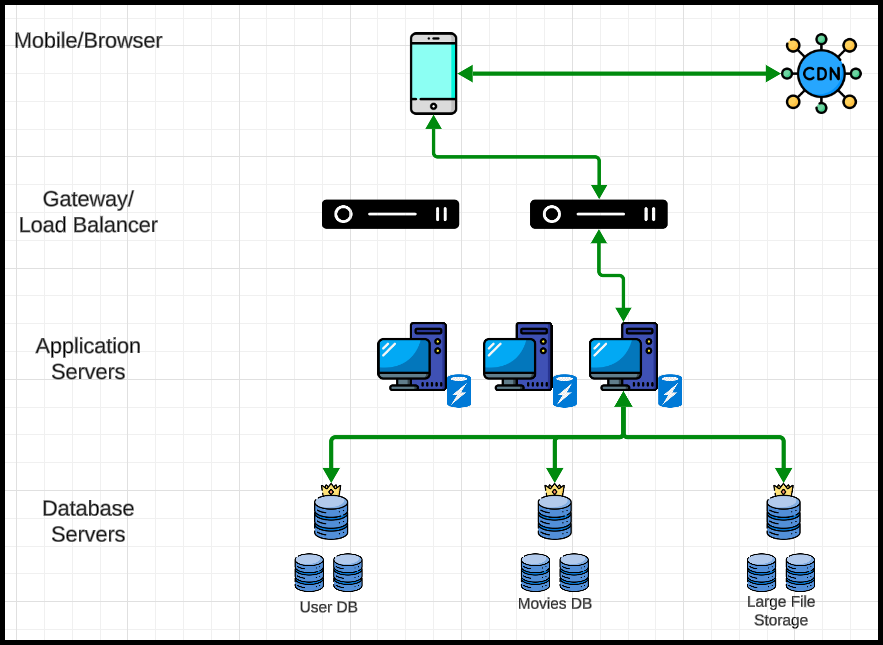
1. **Watch a Movie**
   * Play a movie given a movie ID.
   * User experience spans across browsers, apps, etc.
   * Most frequent use case.
2. **Show List of Movies**
   * Movies watched earlier (recently watched).
   * Watchlist movies.
   * Other lists (e.g., half-watched movies).
3. **Search for Movies or Actors**
   * Ability to search by movie or actor name.

**Excluded Use Cases**

* Standard website features: login/logout, profile management, etc.
* Downloading movies for offline viewing.
* Recommendations like "More Like This."

**Key Concepts and Principles**

1. **Chunking for Efficient Streaming**
   * Movies are stored in chunks to avoid downloading entire files at once.
   * Benefits:
     + Efficient retries (only re-fetch failed chunks).
     + Minimized buffer delays.
   * Chunks are stored in **S3 or HDFS** as the source of truth and **CDN** for faster user access.
2. **CDN Integration**
   * CDN (e.g., Akamai) is used as a caching layer closer to the user.
   * CDN stores chunks fetched from Hotstar's primary storage.
   * CDN URLs for chunks are essential for the browser to retrieve video content.
3. **Metadata**
   * Essential metadata for video playback includes:
     + **General Information:** Title, description, genres, actors.
     + **Playback Information:** Duration, supported audio and subtitle languages, default quality.
     + **Chunk Information:**
       - Start/end timestamps for each chunk.
       - CDN URLs for chunks.
       - Quality/resolution of each chunk.
   * This metadata is lightweight and fetched from Hotstar servers before playback starts.



## Playback Implementation Details

1. **Progressive Downloading and Prefetching**
   * Browser fetches only the first chunk initially.
   * Prefetching of subsequent chunks happens during playback to prevent buffering delays.
   * Prefetching is based on timestamp (e.g., start at 30s if chunks are 60s long).
2. **Seamless User Experience**
   * Smooth playback with low buffering.
   * Dynamic support for seeking (buffering at specific points).
3. **Adaptive Features**
   * Dynamic adaptation to user needs:
     + Quality/Resolution adjustments based on available bandwidth.
     + Previews at specific timestamps.

**Design Questions and Challenges**

**Question 1: How to Implement Preview for a Scene at a Given Timestamp?**

* When the user hovers over the timeline, a snapshot or video preview is shown for the given timestamp.

**Question 2: Limited Storage Handling**

* If a movie is 10 GB but the browser can only store, e.g., 100 MB:
  + How to manage storage efficiently?
  + How to discard old chunks while prefetching new ones?

**Question 3: Auto Resolution and Adaptation**

* How to adjust video quality dynamically?
* Should this only depend on internet speed, or should additional parameters (e.g., device capability) be considered?

## Video Streaming

**1. Preview of a Scene**

* **Chunk Metadata:**
  + Each chunk has:
    - Start and end timestamps, stored as precise UNIX epoch values for synchronization with playback.
    - CDN URL of the video, ensuring fast retrieval via distributed servers.
    - Quality parameter, representing the resolution and bitrate of the chunk.
    - CDN URL of a thumbnail (very low-quality, ~1KB per thumbnail), optimized for minimal storage and network usage.
  + Pre-download thumbnails for all chunks as they are lightweight.
* **Implementation:**
  + Timeline divided into chunks (e.g., 2-minute chunks for a 150-minute movie results in ~70 chunks). This duration strikes a balance between chunk size and efficiency: shorter chunks enhance seek time and reduce buffering for scene navigation, while longer chunks minimize the frequency of network requests, optimizing performance and user experience.
  + Use a function to map a timestamp to a chunk ID:
    - Binary search can be used to find the correct chunk ID efficiently.
  + Fetch and display the corresponding pre-downloaded thumbnail when hovering over a specific timestamp.
* **Behaviour:**
  + As video playback progresses, previews get enabled by downloading thumbnails in parallel.

**2. Managing Large Videos with Limited Storage**

* **Storage Constraints:**
  + Example: 10GB movie split into 100MB chunks; with 200MB available storage, only two chunks can be cached.
* **Caching Strategy:**
  + Store the current and pre-emptively the next chunk.
  + Older chunks are identified and prioritized for removal based on their playback timestamps. Chunks that have already been viewed and are furthest from the current playback position are removed first to manage storage limits efficiently.
* **User Behaviour Impact:**
  + Forwarding within the current or next chunk: no buffering.
  + Rewinding or skipping to older chunks: buffering occurs as older chunks are removed from the cache.
* **Prefetching:**
  + If the browser has sufficient storage, more chunks (e.g., the next 10) can be downloaded sequentially.
  + Pause behaviour: may prefetch additional chunks or the entire video if storage and video size permit.

**3. Resolution Adaptation**

* **Parameters for Resolution Adjustment:**
  + Internet speed.
  + Screen size and aspect ratio.
  + Device type (e.g., mobile, desktop).
  + Data saver mode.
* **Screen Size Adaptation:**
  + Smaller screens (e.g., mobile) may default to lower resolutions (e.g., 720p) to save bandwidth without perceptible quality loss.
* **Internet Speed Monitoring:**
  + An asynchronous background process sends small pings (e.g., 10 bytes) to measure:
    - Time taken for data to travel round-trip.
    - Consistency and health of the connection.
  + Adjust resolution dynamically based on measured speed:
    - Slower speeds trigger downloading lower-resolution chunks pre-emptively.
    - The resolution adjustment process runs every 5 seconds to ensure playback smoothness is maintained. This interval is chosen to balance responsiveness with computational efficiency, ensuring minimal impact on playback performance.
* **Ping Data Consumption:**
  + Minimal data usage (~400 bytes per minute compared to megabytes for video streaming).

**4. Video Quality (Resolution)**

* **Definitions:**
  + Resolution represents the number of pixels in each frame:
    - 1080p: 1920x1080 (2.1 million pixels).
    - 720p: 1280x720 (0.9 million pixels).
    - 4K (4096p): 4096x2160 (8.8 million pixels).
  + Larger resolutions result in better quality but larger file sizes.
* **Aspect Ratio:**
  + Impacts the dimensions and appearance of the video (e.g., horizontal vs. vertical viewing).

**5. Optimization for User Experience**

* **Caching and Buffering:**
  + Proactively download next chunks based on storage and network conditions to minimize buffering. For example, when storage allows, the system may prefetch multiple upcoming chunks, prioritizing smoother playback. The number of chunks downloaded is dynamically determined by evaluating available storage space, current network speed, and anticipated playback patterns.
  + Prioritize forward playback as users are more likely to move forward than backward.
* **Resolution and Bandwidth Management:**
  + Dynamically adjust resolution to balance quality and buffering based on device capabilities and internet conditions.
  + Preserve user experience by avoiding buffering while maintaining acceptable video quality.

**6. Key Takeaways**

* **Thumbnail Previews:** Enable quick scene previews using lightweight thumbnails stored for each chunk.
* **Smart Caching:** Efficient use of limited storage through intelligent chunk management.
* **Adaptive Streaming:** Continuous monitoring of internet speed and device specifications for optimal resolution delivery.
* **User-centric Design:** Balance between quality and performance to enhance overall streaming experience.

## Some Points

**1. Ping and Network Diagnostics:**

* **Ping command**: When you use the ping command from your machine (e.g., pinging www.google.com), it shows:
  + Number of bytes sent
  + Time taken for the ping to travel (round-trip time)
  + Sequence number (this helps track the packets)
  + Time To Live (TTL): Number of hops a packet can take before being discarded.
* **Interpreting ping results**: The lower the ping time (in milliseconds), the better your internet connection is.

**2. Adaptive Bit Rate Streaming (ABS):**

* **Problem in Video Streaming**: When streaming, the internet speed may vary, and if a video starts buffering, it may be due to the internet speed fluctuating during the download of video chunks.
* **Solution**: Adaptive Bit Rate Streaming (ABS) is used, where:
  + A background thread continuously monitors the internet speed (using pings or other methods).
  + Based on the current speed, it decides which resolution of the video to download next.
* **Key idea**: This helps avoid wasting data by ensuring that the video quality matches the available bandwidth.

**3. Downloading Chunks in Video Streaming:**

* Videos are broken down into **chunks** (small video segments), and each chunk can be streamed based on the available bandwidth.
* **Audio and Video Segments**: Just like the video is divided into chunks, the audio is also broken down. For a movie of several hours, each audio file is broken into chunks (e.g., every 2 minutes).
* **Other Metadata**: Each chunk comes with metadata such as:
  + Link to the CDN (Content Delivery Network) URL.
  + Other details that help in fetching and playing the video smoothly.
* **Parallel Threads**: A background thread may be used to prefetch chunks so that buffering is minimized.

**4. Subtitles and Audio Synchronization:**

* **Subtitles**: A subtitle file is a simple text file (often .SRT format) that contains:
  + A timestamp indicating when each subtitle should appear.
  + The text to be displayed at that time.
* **Audio and Video Synchronization**: Just like subtitles, audio is synced to the video based on timestamps.
  + **Resolution**: Video and audio are often broken down into equal-sized chunks (e.g., every 2 minutes).
  + **Timestamp Matching**: The player matches the correct audio with the corresponding video chunk based on the timestamp.

**5. Video Player Design:**

* **Basic Components**: A video player typically handles:
  1. **Video**: The image and changes to the images (visual content).
  2. **Audio**: Audio content, which is synchronized with the video.
  3. **Subtitles**: Text that is synchronized with the video.
* **Efficient Syncing**: As video chunks and audio chunks are fetched and played simultaneously, they must be aligned to avoid delays or out-of-sync issues.

**6. Efficient Subtitle Processing:**

* **Subtitle File Format**: A typical subtitle file (e.g., .SRT) contains:
  + A timestamp (start time and end time) and the text to be shown.
  + The file is usually small (2-5 KB), and subtitles are organized by timestamp in increasing order.
* **Efficient Lookup**: If you were to store subtitles in a **hash map** (with timestamps as keys and subtitles as values), this would allow for constant-time lookup (O(1)) to display the correct subtitle at the right time, making it very efficient for real-time video playback.
* **Alternative Data Structure**: If not using a hash map, subtitles could be stored in an array and processed sequentially in the order of timestamps.

**7. Technical Aspects of Adaptive Streaming:**

* **CDN (Content Delivery Network)**: The video data is fetched from a CDN, which ensures fast and efficient delivery of content from servers closer to the user.
* **Background Thread for Monitoring Speed**: The background thread mentioned earlier checks the download speed and helps decide whether the next video chunk should be downloaded in a lower or higher resolution based on available bandwidth.

**Key Takeaways:**

* **Ping** helps diagnose network issues and measure latency.
* **Adaptive Bit Rate Streaming (ABS)** is crucial for optimizing video quality based on available bandwidth.
* Video and audio in streaming are divided into chunks, with metadata and synchronization mechanisms for smooth playback.
* Subtitles are timestamped and can be stored efficiently in a hash map for quick lookup and alignment with video/audio.
* Efficient streaming and playback require managing video/audio chunks, metadata, and network speed monitoring simultaneously.

## Architecture for Live Streaming

**1. Interactive Live Streaming (Low-Latency, No Rewind/Replay)**

* **Characteristics**:
  + Minimal latency (sub-second to a few seconds).
  + No rewind or replay support during the session.
  + Requires real-time bi-directional communication (e.g., WebRTC).
* **Architectural Adjustments**:
  + Use of **WebRTC** or similar protocols for real-time, low-latency communication.
  + Decentralized peer-to-peer (P2P) architecture for scalability.
  + Media servers (like Janus or Kurento) for handling multiple participants.
  + Prioritization of fast video encoding and minimal buffering.

**2. Broadcast Live Streaming (Higher Latency, With Rewind/Replay)**

* **Characteristics**:
  + Moderate latency (e.g., 2-3 minutes).
  + Viewers can rewind and replay during live streams.
  + Focus on scalability and distribution to millions of users.
* **Architectural Adjustments**:
  + **Chunk-Based Streaming**:
    - Divide the video stream into **chunks** of manageable sizes (e.g., 1-minute chunks).
    - Optimize chunk size to balance latency and overhead.
  + **Encoding Pipeline**:
    - Real-time encoding into multiple resolutions (adaptive bitrate streaming).
    - Parallelized and prioritized processing (e.g., start with low-res chunks like 240p).
    - Efficient transcoding hardware or cloud-based encoding services.
  + **Content Delivery Network (CDN) Integration**:
    - **Edge caching** for faster delivery to viewers.
    - **Pre-fetching strategies** for popular regions/events.
    - Leverage CDNs with low-latency features (e.g., Akamai, Cloudflare).
  + **Buffering and Prefetching**:
    - Clients prefetch the next chunk while playing the current chunk to smooth transitions.
    - Maintain a small playback buffer to handle network variability.
  + **Storage and Metadata**:
    - Maintain metadata tables for chunk resolution and timestamps.
    - Efficiently store and retrieve chunk URLs using databases like DynamoDB or Cassandra.
  + **Client-Server Communication**:
    - Clients poll servers for the next chunk or use push notifications (e.g., WebSocket).
    - Dynamic bitrate adjustments based on client bandwidth.

**Challenges and Optimizations**

1. **Latency Reduction**:
   * Optimize encoding and chunking processes.
   * Use **GPU acceleration** or advanced compression algorithms like AV1.
   * Deploy low-latency CDNs and prioritize nearby edge servers.
2. **Scalability**:
   * Implement adaptive bitrate streaming (ABR) to cater to users with varying bandwidths.
   * Horizontal scaling of encoding and metadata servers.
3. **Resilience**:
   * Build fault-tolerant pipelines for encoding, chunking, and uploading.
   * Redundant servers for handling spikes in traffic.
4. **Enhanced User Experience**:
   * Allow users to switch seamlessly between resolutions.
   * Provide real-time updates on latency or buffer status.

**Interactive vs. Broadcast Live Streaming: A Comparison**

| **Feature** | **Interactive (Zoom/Google Meet)** | **Broadcast (Hotstar)** |
| --- | --- | --- |
| **Latency** | Sub-second to 2 seconds | 2-3 minutes |
| **Rewind/Replay** | Not supported | Supported |
| **Protocol** | WebRTC/RTMP | HLS/DASH |
| **Client Scale** | Dozens to hundreds | Millions |
| **CDN Requirement** | Minimal | Extensive |

## Real-Time Video Architecture and Related Concepts

**1. Real-Time Video Calling Architecture**

* **Centralized Model**:
  + All participants publish and fetch information from a central server.
  + No history is stored—data is relayed directly to subscribers.
  + If an update is missed, it cannot be retrieved later.
  + Centralized systems often use a single machine to handle the meeting.
    - If this machine fails, participants may experience a temporary blip but are quickly shifted to another machine.
* **Peer-to-Peer Model**:
  + In some cases, the server acts as an intermediary to establish direct connections between participants.
  + Example: Google Meet may allow direct exchange of video streams between participants without routing through a central server.
* **Key Characteristics**:
  + Minimal buffering.
  + High reliance on real-time data transfer.
  + Efficient for small-scale, real-time meetings.

**2. Comparison with Content Delivery Platforms like Hotstar**

* **Live Streaming (e.g., Hotstar)**:
  + Designed for large audiences consuming content asynchronously.
  + **Chunking**:
    - Video streams are divided into small time-based chunks (e.g., 1-minute files).
    - Allows continuous transmission to Content Delivery Networks (CDNs).
    - Minimizes lag by promptly transferring smaller data packets.
  + **CDN Usage**:
    - Distributes content globally to reduce latency and improve user experience.
* **Real-Time Calls (e.g., Zoom, Google Meet)**:
  + Data is relayed without storage or chunking.
  + Centralized systems may struggle with scalability if too many publishers (e.g., video feeds) are active simultaneously.
  + Limits on the number of active participants ensure stability.

**3. Live Count in Distributed Environments**

* Clients continuously request the most recent video chunks (polling).
* Servers log these requests to maintain an approximate count of active viewers.
* Example: Logs stored in systems like Elasticsearch are analyzed to calculate live counts based on recent activity.

**4. Recording and Processing Live Video Streams**

* Streams are recorded continuously (e.g., from a stadium).
* The stream is divided into manageable chunks (e.g., every 1 minute).
* Benefits:
  + Faster processing and distribution.
  + Immediate availability of chunks for streaming via CDNs.

**5. Ad Delivery for Different Devices and Languages**

* Ad delivery functions as a separate service with predefined criteria:
  + Target audience parameters (e.g., location, interests, device type).
  + Ads are matched to users based on these parameters before being served.

## Doubt Clearing Session Questions

In case of designing a multi-language OTT platform, the best approach generally involves **separating video and audio files**. Here's why this is often the optimal solution:

**Case 1: Separate Audio and Shared Video**

* **Advantages:**
  1. **Storage Efficiency**: The same video file can be reused across different languages, significantly reducing storage space requirements. Only the audio files change for different languages.
  2. **Reduced Redundancy**: Avoids duplicating video files for each language.
  3. **Flexibility**: Easier to add new languages without duplicating or recreating the video files.
  4. **Synchronization Standards**: Modern video players and streaming protocols are designed to handle separate video and audio tracks seamlessly.
* **Challenges**:
  1. Requires the playback system to synchronize audio with video precisely, but this is a solved problem with existing codecs and media players.
  2. Slight increase in computational overhead on the client device to handle synchronization, but this is negligible on modern devices.

**Case 2: Separate Files for Each Language (Video + Audio Combined)**

* **Advantages**:
  1. No runtime synchronization required, as video and audio are already stitched together.
  2. Slightly less complex for streaming and playback systems.
* **Challenges**:
  1. **High Storage Cost**: Each language requires a full video + audio copy, leading to duplication of large video data.
  2. **Maintenance Overhead**: Managing multiple versions of essentially the same content becomes cumbersome.
  3. **Scaling Issues**: Adding new languages increases storage exponentially compared to just adding audio files.

**Decoding and Playback**

In both cases, during playback:

* **Video and audio are always decoded independently**, even if bundled together in the same file.
* Modern codecs like H.264 and containers like MP4 or MKV allow the separation of audio and video tracks natively.

**Conclusion**

**Case 1 (separate video and audio files)** is better for scalability, storage optimization, and maintainability, especially in a global multi-language scenario. The additional effort to synchronize audio and video during playback is negligible with modern technologies.

1. **Handling High Requests in Live Streaming or Movie Retrieval**

* **Problem:** Many HTTP requests are generated, especially in live streaming, as the browser keeps polling for new chunks.
* **Impact:** Increased demand on the database and backend systems.

1. **Injecting Ads in Live Streams**

* **Techniques for Ad Insertion:**
  + **Superimposing Ads:**
    - Overlay ads on the video while it's playing.
    - Utilize predefined timestamps for inserting ads.
  + **Embedding Ads in Chunks:**
    - Insert ads directly into the video chunks during chunk creation.
    - Best for non-personalized ads (e.g., sports events on platforms like Hotstar).
  + **Comparison with YouTube Ads:**
    - YouTube offers personalized ads, while platforms like Hotstar might have uniform ads for all viewers.

1. **Optimizing Response Time for Authentication**

* **Challenges:** Authentication and description validation for each chunk request in live streaming.
* **Optimization Strategies:**
  + **Caching Session Tokens:**
    - Store session tokens with associated user permissions.
    - Use the cache to validate user access quickly.
  + **Avoiding Redundant Checks:** Reduce repeated database queries.

1. **Handling Fast-Forwarding in Videos**

* **Issue:** Jumping to a different timestamp without preloaded chunks causes buffering.
* **Solution:**
  + Perform a **binary search** to find the chunk corresponding to the desired timestamp.
  + Use the chunk’s **CDN URL** to fetch and play it.

1. **Securing CDN Links**

* **Potential Risks:**
  + Browser storage can expose CDN links.
  + Users might save links for unauthorized future use.
* **Mitigation Techniques:**
  + Set a **TTL (Time-To-Live)** for CDN URLs.
  + Regularly refresh and re-upload content to invalidate old URLs.

1. **Sharing Content Legally and Illegally**

* **Legal Concerns:**
  + Sharing screens via tools like Google Meet allows unauthorized viewing.
  + Though illegal, such actions are difficult to prevent entirely.