

MEDIA STREAMING IBM AND CLOUD VIDEO STREAMING BATCH MEMBERS

Vibhu vishwa deep.B.K

Vinoth kumar.K

Srihari sundhar.S

Vigneshwaran.T

INTRODUCTION:

Streaming is a method of viewing video or listening to audio content without actually downloading the media files.

Streaming performance can be improved, and buffering time reduced, if the owner of the files uses a CDN.

Overview of Stream Data Processing

Today's data is generated by an infinite amount of sources - IoT sensors, servers, security logs, applications, or internal/external systems. It's almost impossible to regulate structure, data integrity, or control the volume or velocity of the data generated.

While traditional solutions are built to ingest, process, and structure data before it can be acted upon, streaming



data architecture adds the ability to consume, persist to storage, enrich, and analyze data in motion.


As such, applications working with data streams will always require two main functions: storage and processing. Storage must be able to record large streams of data in a way that is sequential and consistent. Processing must be able to interact with storage, consume, analyze and run computation on the data.

This also brings up additional challenges and considerations when working with legacy databases or systems. Many platforms and tools are now available to help companies build streaming data applications.

Examples

Some real-life examples of streaming data include use cases in every industry, including real-time stock trades, up-to-the-minute retail inventory management, social media feeds, multiplayer games, and ride-sharing apps.

For example, when a passenger calls Lyft, real-time streams of data join together to create a seamless user experience. Through this data, the application pieces together real-time location tracking, traffic stats, pricing, and real-time traffic data to simultaneously match the rider with the best possible driver, calculate pricing, and estimate time to destination based on both real-time and historical data.



In this sense, streaming data is the first step for any data-driven organization, fueling big data ingestion, integration, and real-time analytics.

Batch Processing vs Real-Time Streams

Batch processing requires data to be downloaded as batches before it can be actionable, whereas streaming data allows for simultaneous, real-time processing, storage, and analytics.

With the complexity of today's modern requirements, legacy batch data processing has become insufficient for most use cases, as it can only process data as groups of transactions collected over time. Modern organizations need to act on up-to-the-millisecond data, before the data becomes stale. Being able to access data in real-time comes with numerous advantages and use case

•

BASIC EXAMPLE:

```
const videoEl = document.querySelector('video');
const mediaSource = new MediaSource();

video.src = URL.createObjectURL(mediaSource);
mediaSource.addEventListener(
  'sourceopen',
  () => {
    const mimeType = 'video/mp4; codecs="avc1.42E01E, mp4a.40.2"';
    const buffer = mediaSource.addSourceBuffer(mimeType);

    buffer.appendBuffer( /* Video data as `ArrayBuffer` object. */ )
  }
);
```

Part 1: Setup project

You'll need to install NodeJS and run:

```
mkdir http-video-stream
cd http-video-stream
npm init
npm install --save express nodemon
```

Part 2: index.html

We need to create a HTML5 Video element, and set the source as `/video`, which is where server's endpoint is.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>HTTP Video Stream</title>
  </head>
  <body>
    <video id="videoPlayer" width="650" controls muted="muted" autoplay>
      <source src="/video" type="video/mp4" />
    </video>
  </body>
</html>
```

Part 3: index.js

Now lets setup our node server so that on `/` endpoint it serves our index.html page.

```
const express = require("express");
```



```
const app = express();

app.get("/", function (req, res) {
  res.sendFile(__dirname + "/index.html");
});

app.listen(8000, function () {
  console.log("Listening on port 8000!");
});
```

Part 4: package.json -- Run our server

Add a `start` script to your `package.json` so that we can run our server using `npm start` command.

There's more in your `package.json` file but I just want you to copy this `start` script. It uses `nodemon` to run `index.js` and restarts the server every time you save the `index.js` file so you don't need to restart the server yourself!

```
{
  "scripts": {
    "start": "nodemon index.js"
  }
}
```

Now you should be able to run

`npm start`

and see our app running on port 8000. Open your browser and go to `http://localhost:8000` to see if it worked.

Part 5: index.js (Again)

We're almost there!

Here's the `/video` endpoint for our server.

```
// in the imports above  
const fs = require("fs");
```

```
app.get("/video", function (req, res) {  
  // Ensure there is a range given for the video  
  const range = req.headers.range;  
  if (!range) {  
    res.status(400).send("Requires Range header");  
  }  
}
```

```
// get video stats (about 61MB)  
const videoPath = "bigbuck.mp4";  
const videoSize = fs.statSync("bigbuck.mp4").size;
```

```
// Parse Range  
// Example: "bytes=32324-"  
const CHUNK_SIZE = 10 ** 6; // 1MB  
const start = Number(range.replace(/\D/g, ""));  
const end = Math.min(start + CHUNK_SIZE, videoSize - 1);
```

```
// Create headers  
const contentLength = end - start + 1;  
const headers = {  
  "Content-Range": `bytes ${start}-${end}/${videoSize}`,  
  "Accept-Ranges": "bytes",  
  "Content-Length": contentLength,  
  "Content-Type": "video/mp4",  
};
```

```
// HTTP Status 206 for Partial Content
```

```
res.writeHead(206, headers);  
  
// create video read stream for this particular chunk  
const videoStream = fs.createReadStream(videoPath, { start, end });  
  
// Stream the video chunk to the client  
videoStream.pipe(res);  
});
```

The HTML5 video element makes a request to the `/video` endpoint, and the server returns a file stream of the video, along with headers to tell which part of the video we're sending over.

CONCLUSION:

- **Large volumes of continuous data, possibly infinite.**
- **Steady changing and requires a fast, real-time response.**
- **Data stream captures nicely our data processing needs of today.**