How Spotify Streams to 700M Users Without Skipping a Beat

SPOTIFY

- Launched in 2008 by a Stockholm-based startup.
- Revolutionized the music industry, surpassing 750 million active users.
- Library of 100M+ songs, 7M podcasts, and 500,000+ audiobooks.
- Empowers millions of independent creators with global distribution and real-time listener data.

FUNCTIONAL REQUIREMENTS

- Users can maintain profiles.
- Users can create playlists.
- Users can search for songs, albums, artists, and playlists by keyword.
- Users can stream songs in real time from anywhere in the world.
- Users can receive personalized recommendations from the search history.

NON-FUNCTIONAL REQUIREMENTS

- The system should handle millions of users globally with the ability to stream millions of songs concurrently.
- Real-time streaming must have low latency for a seamless user experience.
- The system must be available at all times with minimal downtime.
- Support users from different geographic regions.

Assumptions & Scale

User assumptions

- Monthly Active Users (MAU): ~ 700 million worldwide.
- DAU : ~ 300 million
- Premium Subscribers: 290 million.
- Artists: Over 10 million.

Content assumptions

- Playlist : ~ 800M
- Albums : 8 albums per artist ~ 80 million

Track assumptions

- Tracks : ~100 million
- Average streams per user per day: 10
- Average song size: 5 MBs
- Average song duration: 4 minutes
- Average Number of Songs Posted per day : 100k

Read & Write QPS

Read Qps

Streams/user/day = 10

- 300M×10 = 3,000M=3 billion requests/ day
- 3,000M /86,400 = 34,722 qps

Search/user/day ~ 5

- 300M × 5 = 1.5B≈ ~17K qps

Recommendations/user/day ~3

- 300M × 3 = 900M≈ ~10K qps

Total Read QPS ~ 62K
Peak : 62,000 × 2 ~ 124k

Write Qps

Profile edits/user/day

- ~5% of DAU edit daily → 15M
- 15,000,000 / 86,400 ≈ 174

Create/update playlists ~ 3M

- 30,000,000 / 86,400≈ 347

Log search history ~ 5

- 300M × 5 = 1.5B ~17K

Log play events ~ 10

- 300M × 10 = 3B ~35K

Song uploads/ day : 100k
100K≈ ~1

Total Write QPS~52K
Peak : 52,605 × 2 ~ 105K

Storage

PostgreSQL : For read-heavy complex joins.
Cassandra: For High-volume append-heavy writes.
Elastic search: For full-text keyword search.
Redis : Millisecond reads for hot cached data.
S3: Blob storage for large audio files.

Entity	Database
Album	PostgreSQL
playlists	Cassandra
Play Events (log)	Cassandra
Search History (log)	Cassandra
Search Engine	Elastic Search
Cache	Redis

Entity	Database
User	PostgreSQL
playlists	PostgreSQL
Artist	PostgreSQL
Track Metadata	PostgreSQL
Track Audio files	Object Storage (S3)

Storage Requirements

Row Size Calculation

- ***Users*** : users 4 (user_id) + 50 (name) + 50 (email) + 50 (country) + 50 (subscription_type) + 4 (created_at) = 208 bytes
- ***Song***: song_metadata 4 (song_id) + 50 (title) + 4 (artist_id) + 4 (album_id) + 50 (genre) + 100 (file_location) + 4 (duration) + 4 (release_date) = 220 bytes
- ***Playlist*** : playlists 4 (playlist_id) + 4 (user_id) + 50 (playlist_name) + 100 (song_ids) + 4 (created_at) = 162 bytes
- ***Artists***: artists 4 (artist_id) + 50 (name) + 200 (bio text) + 4 (debut_year) = 258 bytes
- ***Albums***: albums 4 (album_id) + 50 (title) + 4 (artist_id) + 4 (release_date) = 62 bytes

Table Size Calculation

- ***Users:*** 208 × 700M = 145.6 GB
- ***Song :*** 220 × 100M = 22 GB
- ***Albums :*** 62 × 80M = 5 GB
- ***Playlist :***162 × 4,000M = 648 GB
- ***Artists :*** 258 × 10M = 2.6 GB
- ***Total~824 GB ≈ ~0.8 TB***

Growth Estimates

Assuming 10% growth each year

Year	Users	Paid Users	Tracks	Artists	Playlists
Year 0 (Now)	700M	290M	100M	10M	8000M
Year 1	770M	319M	115M	13M	8960M
Year 2	847M	351M	132M	16.9M	10035M
Year 3	932M	386M	152M	22M	11240M
Year 4	1025M	425M	175M	28.6M	12589M
Year 5	1128M	468M	201M	37.1M	14100M

Growth Estimates

Assuming 10% growth each year

Entity	Year 1	Year 2	Year 3	Year 4	Year 5
Users	156.2 GB	171.8 GB	189.0 GB	207.9 GB	228.7 GB
Songs	22.0 GB	24.2 GB	26.6 GB	29.3 GB	32.2 GB
Playlists	1296.0 GB	1425.6 GB	1568.2 GB	1725.0 GB	1897.5 GB
Artists	2.6 GB	2.8 GB	3.1 GB	3.4 GB	3.8 GB
Albums	3.1 GB	3.4 GB	3.8 GB	4.1 GB	4.5 GB
Total	1479.9 GB	1627.9 GB	1790.7 GB	1969.7 GB	2166.7 GB

API'S

Recommendation

GET /recommendations/{user_id}

Request:

Authorization:<jwt_token>

Response:


```
{
  "recommendations": [
    {
      "song_id": "10234",
      "title": "Sky full of stars",
      "artist": "Coldplay",
      "score": 0.98
    },
    {
      "song_id": "67890",
      "title": "When we feel young",
      "artist": "Chai met toast",
      "score": 0.95
    }
  ]
}
```


Search

GET /search?q=sky+full+of+stars&type=song

Request:

Authorization:<jwt_token>

Response:


```
{
  "query": "sky full of stars",
  "type": "song",
  "total": 2,
  "results": [
    {
      "song_id": 10234,
      "title": "A Sky Full of Stars",
      "artist": "Coldplay",
      "album": "Ghost Stories",
      "duration_ms": 267893,
      "release_year": 2014
    },
    {
      "song_id": 11042,
      "title": "A Sky Full of Stars (Remix)",
      "artist": "Coldplay ft. Avicii",
      "album": "Ghost Stories Remixed",
      "duration_ms": 298000,
      "release_year": 2014
    }
  ]
}
```


Track streaming

GET /songs/{id}

Request:

Authorization:<jwt_token>

Response:


```
{
  "song_id": 10234,
  "title": "sky full of stars"",
  "artist": "Coldplay",
  "duration_ms": 200040,
  "signed_url": "https://
cdn.spotify.com/audio/1024?
token=xyz"
}
```

Track Streaming Flow

GET /stream/{song_id}

Request:

Authorization: Bearer <jwt_token>

Range: bytes=0-524287

Response:

Status: 206 Partial Content

Content-Type: audio/mpeg

Content-Range: bytes 0-524287/10485760

Body: <binary audio chunk>

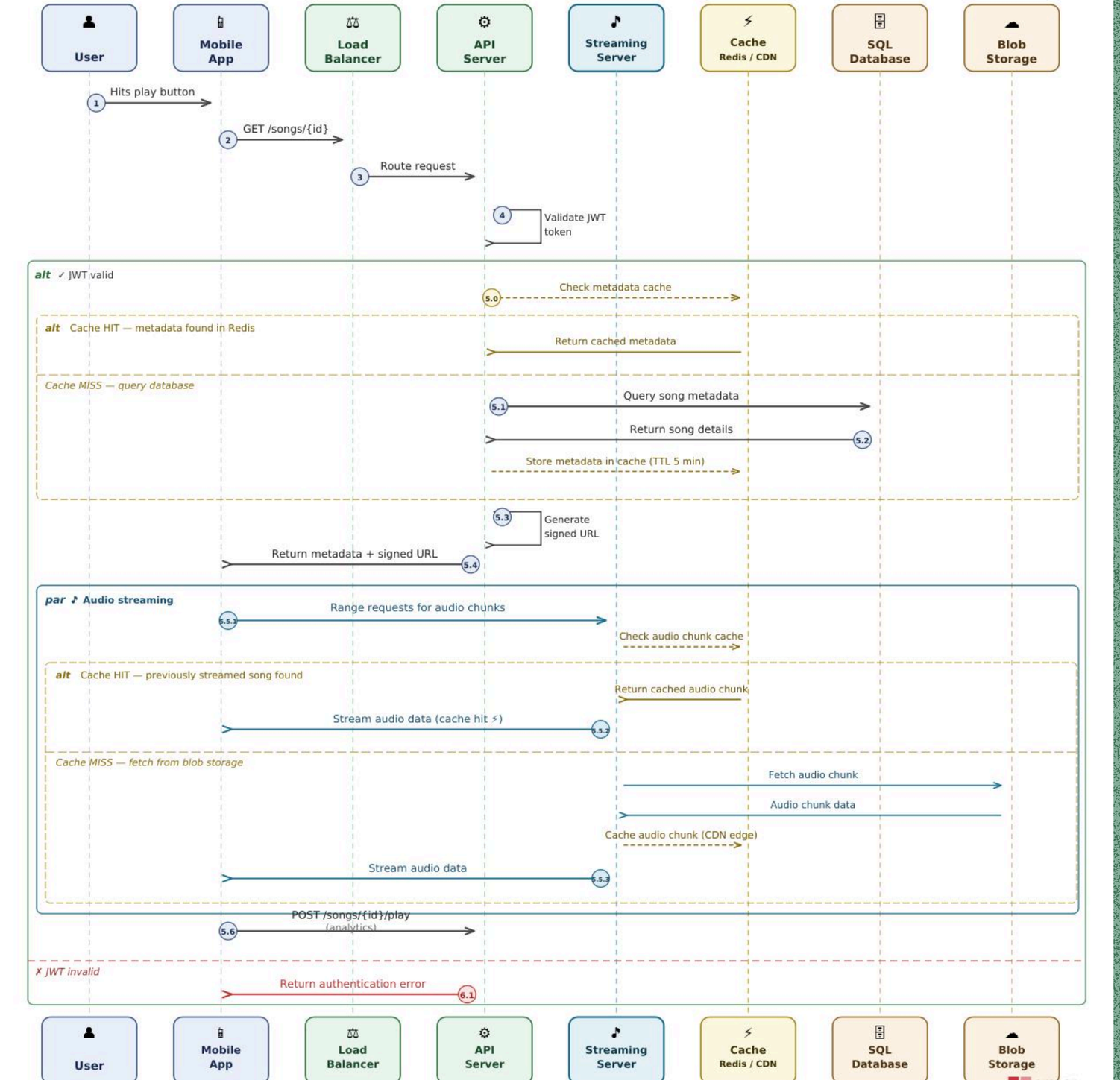
POST /songs/{id}/play

Request:

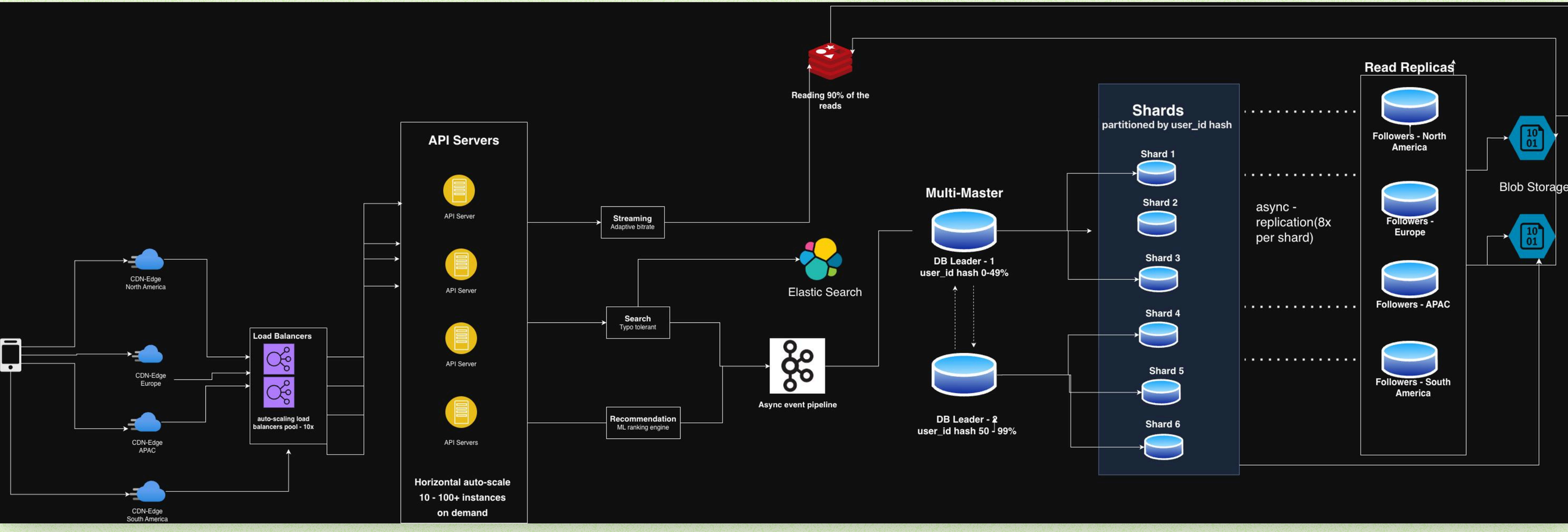
```
{
  "song_id": 10234,
  "user_id": 99012,
  "played_at": "2024-04-04T10:32:00Z",
  "duration_played_ms": 200040
}
```

Response:

```
{
  "status": "recorded"
}
```



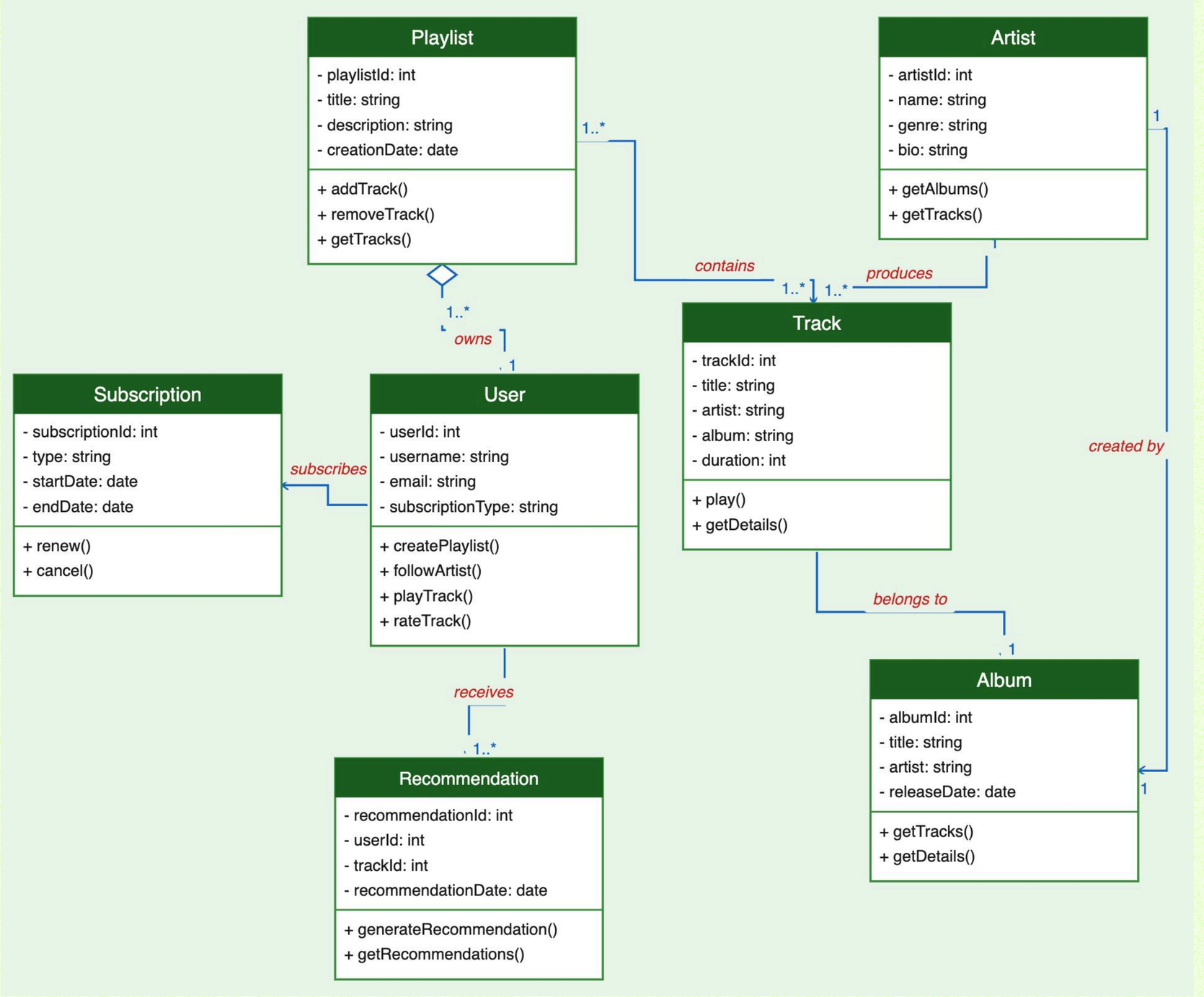
HLD


Reading 90% of the reads
API Servers
API Server
API Server
API Server
API Servers
Horizontal auto-scale
10 - 100+ instances
on demand
Streaming
Adaptive bitrate
Search
Typo tolerant
Recommendation
ML ranking engine
Elastic Search
Async event pipeline
Multi-Master
DB Leader - 1
user_id hash 0-49%
DB Leader - 2
user_id hash 50 - 99%
Shards
partitioned by user_id hash
Shard 1
Shard 2
Shard 3
Shard 4
Shard 5
Shard 6
async - replication(8x per shard)
Read Replicas
Followers - North America
Followers - Europe
Followers - APAC
Followers - South America
Blob Storage
CDN-Edge North America
CDN-Edge Europe
CDN-Edge APAC
CDN-Edge South America
Load Balancers
auto-scaling load balancers pool - 10x

ERD Diagram

Entities

- Playlist
- Artist
- Track
- User
- Subscription
- Album
- Recommend ation



Recommendation Engine Architecture

3 Techniques (run in parallel):

- Collaborative Filtering → Finds users like you → recommends what they enjoy
- NLP (Text Analysis) → Analyzes blogs & playlists → understands cultural context
- Audio Analysis → Processes song signals → extracts tempo, mood, energy.

Each one has blind spots the others cover.

How this scales?

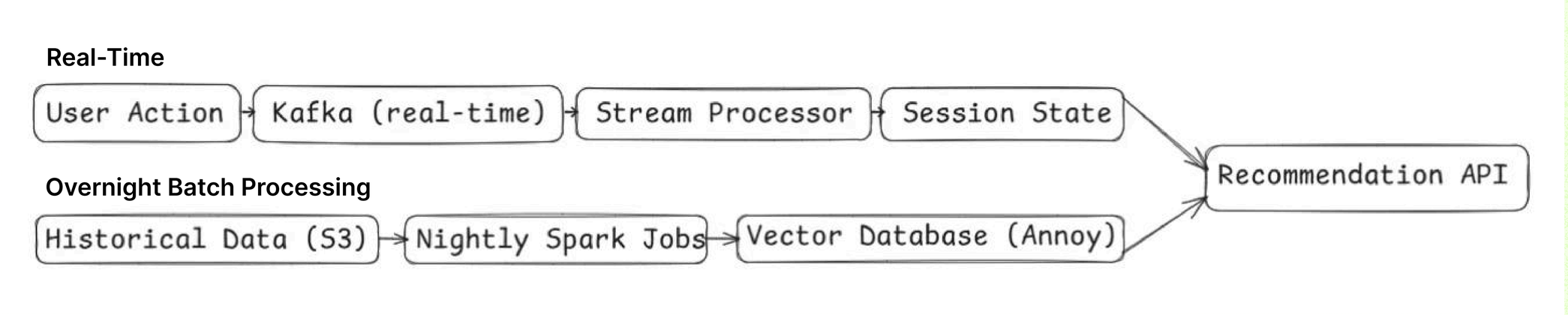
Originally: 700 million × 100 million cells(songs) ~ 50 petabytes of RAM.

Using **Matrix Factorization** for computational and storage scalability.

```
user_vector = [0.8, 0.3, 0.9, ...]  # 200 taste factors
song_vector = [0.7, 0.4, 0.8, ...]  # same 200 factors

similarity = dot_product(user_vector, song_vector)
High similarity = good recommendation
```

Compress billions of interactions into small vectors



Key Decisions

- Cassandra can handle 1M+ writes/sec per node making it highly suitable for song play write and every search query logged for recommendation.
- Elastic search cluster
  - Support typo-tolerant search across 200M songs in <100ms
  - Inverted index — maps every word to all songs containing it.
  - Lookup is O(1)
  - Handles 10,000+ search QPS per node.
- To communicate between 3 services, using Kafka as it can handle 1M+ msg/sec per broker(rarely overwhelmed).
- CDN edge servers for audio from <50ms away.
- Redis will handle 100,000 QPS per node. At 62,000 read QPS, a single Redis node can handle the entire load.
- PostgreSQL sharding: User data is horizontally sharded by user_id hash, ensuring every query routes to exactly one shard without a full table scan.

Thank you!