Evolution Digraph Database for Lossless Audio Data Storage + Efficient Streaming

Isaac Leandro Penzol-Kenyon
Neoko LLC, Iowa, USA

May 4, 2026

Abstract

The first recorded sound was in 1857[1]. Current Americans spend almost 20% of every day with audio[2]. Of those same Americans, anyone 13+ now spends an average of 18% of each audio day listening to streamed music[3]. The following is just the equation of how many hours H per day are used by population P given usage per day U.

$$H = P \cdot U$$

Figure 1: Simple usage estimation for a population

Then we assume an American population of 350,000,000 and given 4.8 hours per day of usage from the above percentages.

1,680,000,000 hours of audio streamed every day.

70,000,000 hours of audio streamed every hour.

1,166,666 hours of audio streamed every minute.

19,444 hours of audio streamed every second.

19 hours of audio streamed every millisecond.

Figure 2: Estimated audio usage in the United States.

This opens your eyes to just the sheer size, output, and infrastructure of todays streaming, storage, software, and hardware behind the computer systems that make it possible.

1 Introduction

Graph databases have been around since the mid-1960s[4] and the design of those graph databases has been improving ever since. In this paper, some related work on past designs will be mentioned. The main problem statement will be addressed. A high-level design made for audio will be introduced. A conceptual implementation. And evaluations + limitations were measured directly from a live commercial product www.neokomusic.com.

2 Problem Statement

How can a storage system for audio file access be optimized so that the most number of people can stream and store the audio data at one time?

2.1 Questions:

Q_1: Why does persistent storage of a graphs current state matter?
Q_2: What can this design be extended to?
Q_3: Why audio as an example?

2.2 Answers:

A_1: Not having to read in large amounts of data to setup the graph state each time a request is made or server is rebooted, useful for large data centers and active graph research into extremely large graph sets. Persistent storage still also allows quick access to any node in the graph.
A_2: Any type of file based storage.
A_3: Audio is both culturally ubiquitous, computationally expensive to store, and hard to diff.

3 Related Work

3.1 Current Designs

There are a variety of different types of graph storage in both current research and commercial applications. Such as adjacency list, edge table, property graph, RDF triple, native disk-based engines, distributed, and temporal / versioned.

3.2 Presented Design

The presented design uses temporal + distributed + edge table approaches as the three main design pillars. The design tries to use these 3 approaches simultaneously to improve speed and efficiency.

4 Design

4.1 Evolution Digraphs

Evolution digraphs are a tree graph where each child node can only be pointed towards its parent node. The parent node can have any number n of children, but those children can have only one parent each.

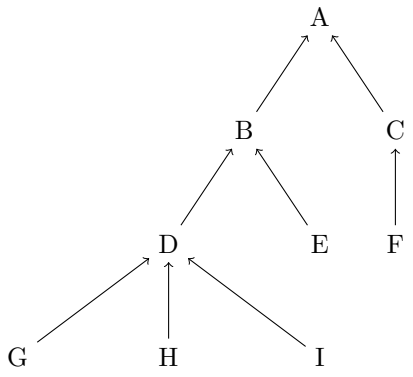

Figure 3: Example diagram of an evolution digraph

This graph gives very interesting properties, such as from any node inside of graph G there is only one path to the root node. It also means that in terms of data, each node only needs to hold one link to a parent. Rather than the parent node holding all the childrens data. Because trees have fast

root node path search times, navigating up and down the tree also becomes quick. Enabling real-time views of extremely large amounts of evolution data stored. This in turn could be used for complete file versioning compared to git[5] which was designed for diff file versioning.

4.2 Finding a Node

Finding a node then becomes as easy as each node is a unique primary key that is searchable in log(n) time. Finding a parent node becomes log(n) time. Recently accessed nodes are then cached in a large RAM memory to speed up common queries. Responding well to real user interactions.

4.3 What is Tracked in Each Node

Each node then has an i number of audio files. These audio files may be represented as completed tracks or stems that make up an entire song.

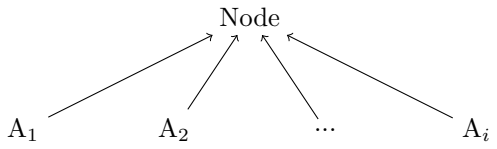

Figure 4: Audio files attached to a chosen node

4.4 Audio Versioning – And Why it's Hard!

It is impossible for all audio files to be broken down where from a source input, there is a linear transformation output. This is why audio is generally a file type that needs full file versioning.

Figure 5: Diagram showing audio processing pipelines.

EQ and Compression can both be non-linear transforms that reshape the entire audio file. Thus every version of an audio file must be saved, growing the size of storage quickly. The evolution digraph is efficient for navigating these changes.

5 Implementation

5.1 Hardware

A Dell PowerEdge R730 server was used with an SSD memory card acting as the boot and storage device.

5.2 Internet Speed

The system was tested on a 100 Mpbs data connection.

5.3 Operating System

The FreeBSD operating system is used with the ZFS filesystem as it is specialy designed for large file storage. The operating system comes with pkgs or ports that can install all the needed software. FreeBSD also has efficient networking capabilities that is useful for the amount of audio needing to be streamed.

5.4 Codebase Structure

The codebase is developed in a horizontally scaled monolith way. This allows the monolithic code to be placed behind a load balancer.

5.5 HTTP Server

Nginx is used to serve and cache static files, as a reverse proxy, and also load balancing. Nginx also works well to secure the server, blocking off any unauthorized directories in the HTTP request.

5.6 Audio Database

A MySQL database is used to store each node. There exists a node table with the columns: node, parent, database. Each node either points to a parent node or is the root. The database column represents where the current data is stored in the physical world i.e. server 1, 2, 3, etc. The database column is useful for streaming from multiple servers and tracking redundancy.

5.7 Web Server

The Python Flask framework is used for all node CRUD actions. The python code then maps the location of the database, and interacts with MySQL to make a query.

5.8 Front End

Vanilla Javascript is used as the client side processing language. It allows for yet another caching layer at the client. All requests made to the server API are cached with a version key and TTL. The front end is designed to be the most aggressive cache possible as server can update version keys anytime.

5.9 Scripts

Sh scripting is used as it comes native to FreeBSD to be the program start and test CI/CD pipeline. The sh scripts are used to manipulate, read, and analyze log files, create backups, and called in cron jobs.

6 Evaluation

The script shown below is to test the endpoints. And the resulting tables help gain insights into how the system performs under different condition.

```
#!/bin/bash

AUDIO_URL="https://neokomusic.com/api/audio/(uuid)"
RANDOM_URL="https://neokomusic.com/random"
DURATION="30s"

THREADS=(1 2 4 8 16 32 64)
CONNS=(1 10 100 1000 10000 100000)

# thread and connection ranges
THREADS_2=(1 2 4 8 16)
CONNS_2=(1 10 50 100 250 500 1000 2500 5000 10000)

mkdir -p wrk_logs_RANDOM_2
mkdir -p wrk_logs_AUDIO_2

for t in "${THREADS[@]}"; do
  for c in "${CONNS[@]}"; do

    echo "Running wrk: threads=$t connections=$c"

    wrk -t$t -c$c -d$DURATION "$RANDOM_URL" \
      > "wrk_logs_RANDOM/t${t}_c${c}_d30s.log"

    wrk -t$t -c$c -d$DURATION "$AUDIO_URL" \
      > "wrk_logs_AUDIO/t${t}_c${c}_d30s.log"

  done
done

for t in "${THREADS_2[@]}"; do
  for c in "${CONNS_2[@]}"; do

    echo "Running wrk: threads=$t connections=$c"

    wrk -t$t -c$c -d$DURATION "$RANDOM_URL" \
      > "wrk_logs_RANDOM_2/t${t}_c${c}_d30s.log"

    wrk -t$t -c$c -d$DURATION "$AUDIO_URL" \
      > "wrk_logs_AUDIO_2/t${t}_c${c}_d30s.log"

  done
done
```

Figure 6: Script to test number of threads and connections.

Threads	Conns	Req/s	Avg Latency	Timeouts	Non-2xx
1	100000	187.14	1.10 s	1130	5627
1	10000	256.40	894.91 ms	2181	7713
1	1000	176.33	1.05 s	1162	5303
1	100	0.20	78.51 ms	0	6
1	10	1.80	1.27 s	52	0
1	1	1.43	678.01 ms	0	43
2	100000	195.18	1.18 s	1167	5874
2	10000	195.42	1.16 s	962	5882
2	1000	297.29	828.56 ms	623	8929
2	100	0.33	86.86 ms	1	9
2	10	1.89	785.78 ms	52	3 ng
4	100000	922.89	397.80 ms	258	27740
4	10000	999.36	512.16 ms	884	30085
4	1000	599.68	566.47 ms	1046	18042
4	100	0.20	130.19 ms	0	6
4	10	1.73	1.90 s	51	0
8	10000	332.32	778.46 ms	1953	9999
8	1000	427.25	608.37 ms	1276	12844
8	100	0.10	52.63 ms	0	3
8	10	1.59	1.39 s	47	0
16	1000	1248.59	453.70 ms	369	37583
16	100	0.40	174.93 ms	1	10
32	1000	166.87	1.03 s	496	5023
32	100	0.20	48.02 ms	1	5
64	1000	148.68	954.78 ms	810	4472
64	100	2.26	57.86 ms	64	4

Table 1: wrk benchmark results for `/audio/(uuid)` endpoint

Threads	Conns	Req/s	Avg Latency	Timeouts	Non-2xx
1	10000	66.25	1.83s	1855	1993
1	1000	84.21	999.87ms	1094	2531
1	100	0.30	267.23ms	0	9
1	10	1.43	553.64ms	39	0
1	1	1.43	712.57ms	0	0
1	2500	58.49	1.86s	1437	1761
1	250	1.99	291.29ms	0	60
1	5000	48.04	1.84s	1424	1445
1	500	13.02	551.61ms	8	390
2	10000	346.76	976.80ms	497	10426
2	1000	125.44	979.52ms	1658	3777
2	100	45.16	157.90ms	0	1359
2	2500	102.38	1.31s	1595	3078
2	250	3.86	1.19s	1	116
2	5000	240.82	671.96ms	1869	7234
2	500	22.11	1.35s	306	664
2	50	0.00	–	0	0
4	10000	191.20	1.18s	1596	5752
4	1000	307.81	792.38ms	611	9265
4	100	4.32	62.79ms	1	129
4	10	2.03	1.35s	59	0
4	2500	229.46	877.18ms	1611	6906
4	250	11.40	149.21ms	0	343
4	5000	290.61	864.68ms	1398	8735
4	500	17.40	486.81ms	4	523
4	50	1.76	49.32ms	48	5
8	10000	248.38	872.94ms	1461	7476
8	1000	171.40	1.08s	757	5158
8	100	0.10	274.28ms	0	3
8	10	2.00	1.64s	56	0
8	2500	261.89	862.82ms	1195	7877
8	250	71.91	96.35ms	9	2162
8	5000	814.63	578.98ms	1175	24507
8	500	61.69	378.13ms	202	1857
8	50	1.83	43.91ms	49	6
16	1000	153.61	1.08s	1249	4623
16	100	0.10	163.07ms	0	3
16	250	0.60	728.24ms	0	18
16	500	8.21	423.89ms	49	247

Table 2: wrk benchmark results for `/audio/(uuid)` endpoint

Threads	Conns	Req/s	Avg Latency	Timeouts	Non-2xx
1	1	11.55	195.21 ms	0	0
1	10	89.67	200.10 ms	0	0
1	100	327.03	231.92 ms	305	994
1	1000	2797.43	293.11 ms	1202	82788
1	10000	2662.07	300.65 ms	1094	79029
1	100000	2362.07	310.01 ms	959	70096
2	10	96.55	224.62 ms	0	129
2	100	439.05	230.43 ms	136	1122
2	1000	3238.47	267.85 ms	1072	96116
2	10000	2683.94	310.33 ms	1192	79586
2	100000	3122.65	262.96 ms	1316	92338
4	10	66.19	196.67 ms	0	0
4	100	340.12	278.50 ms	166	1003
4	1000	2501.94	339.75 ms	1246	74221
4	10000	2787.71	304.17 ms	1144	82400
4	100000	2903.98	282.49 ms	1234	86252
8	10	77.19	170.63 ms	0	1
8	100	620.77	226.36 ms	0	1327
8	1000	2545.42	337.35 ms	1195	75522
8	10000	2931.12	288.06 ms	980	87124
16	100	528.54	231.60 ms	0	1013
16	1000	2907.24	306.40 ms	1279	85906
32	100	360.62	241.89 ms	177	869
32	1000	2554.54	336.33 ms	1224	76032
64	100	490.48	200.36 ms	0	193
64	1000	2653.50	308.91 ms	1248	78567

Table 3: wrk benchmark results for `/random` endpoint

Threads	Conns	Req/s	Avg Latency	Timeouts	Non-2xx
1	1	7.35	189.07 ms	0	0
1	10	97.13	221.50 ms	0	0
1	50	439.22	227.18 ms	0	51
1	100	354.32	301.12 ms	210	936
1	250	2195.26	242.64 ms	535	63536
1	500	3123.57	187.00 ms	842	92272
1	1000	2644.36	310.77 ms	1204	78502
1	2500	2338.81	339.80 ms	1141	69164
1	5000	2115.26	382.14 ms	1075	62379
1	10000	2919.46	277.52 ms	1058	86831
2	10	66.76	323.06 ms	0	0
2	50	237.47	252.70 ms	0	15
2	100	602.23	237.21 ms	0	1433
2	250	2197.35	233.04 ms	540	63360
2	500	3291.93	217.30 ms	782	96931
2	1000	3136.33	282.74 ms	1247	93138
2	2500	2977.58	280.97 ms	1231	88433
2	5000	2198.70	378.98 ms	858	65319
2	10000	2835.90	286.76 ms	1203	84045
4	10	61.11	229.97 ms	0	0
4	50	385.14	213.65 ms	0	36
4	100	414.85	250.18 ms	159	1149
4	250	2210.23	228.68 ms	695	64464
4	500	3346.66	184.30 ms	950	98754
4	1000	2778.36	319.75 ms	1169	82344
4	2500	2862.91	293.95 ms	1004	85029
4	5000	2617.75	317.50 ms	1227	77874
4	10000	2450.12	345.22 ms	827	72860
8	10	23.10	321.54 ms	0	0
8	50	429.83	224.31 ms	0	55
8	100	368.24	258.77 ms	131	759
8	250	2233.31	240.53 ms	505	64828
8	500	3305.28	152.96 ms	1246	97669
8	1000	2620.38	334.25 ms	1225	77313
8	2500	2450.78	333.30 ms	981	73016
8	5000	2585.84	322.41 ms	452	77540
8	10000	2757.81	300.24 ms	1040	81982
16	50	398.23	210.34 ms	0	46
16	100	583.77	282.09 ms	7	1120
16	250	2185.65	238.78 ms	547	63528
16	500	3223.71	213.14 ms	727	95112
16	1000	2676.38	319.32 ms	1150	79585

Table 4: wrk benchmark results for `/random` endpoint

7 Limitations

7.1 Hardware Used

How would this design scale to a much larger system? How can modern hardware speed it up?

7.2 Internet Speed

This system is running on 100Mbps upload/download. How can higher internet speed enable larger throughput?

7.3 Improvements to Streaming

The system streaming does work with one or a few active users. So for research, large systems are possible. Seek time streaming for audio was not implemented, instead a compressed whole file is uploaded/downloaded. Can further compression and efficient streaming algorithms be used to increase the number of nodes that can be accessed per user?

8 Conclusion

The design has real-world usability and is currently being used within a live product. The system is stable and can run without interruptions and the search is in log(n). Fields that may use graphs to model a system can benefit from the design if each node needs to hold file data.

8.1 Future

150+ drives were acquired to upgrade the storage capacity and grow the number of distributed servers. The goal is to progressively build out the amount of storage as needed. Continue to improve the software optimizations. Start thinking about customized hardware. Look into collecting data for every endpoint and API

Acknowledgments

This work was supported by Neoko LLC. The implementation was developed as part of ongoing commercial research and development.

Figure 7: "NO DATA" - An example of how easy it is to make a drive unreadable

Figure 8: Version 1 hardware of the digraph database distributed system running

Figure 9: Version 2 hardware, Dell PowerEdge R730(What was used for all measurments)

Figure 10: 150+, 1TB+ drives for future builds

Figure 11: Closer inspection of the 150+ drives

References

[1] Library of Congress. *Recorded Sound Timeline*. 2013.

[2] Nielsen. *The Record: Q2 U.S. Audio Listening Trends*. 2024.

[3] Daniella Peter Paul-Loor. *Substantial Shifts in the Audio Day of Americans*. 2024.

[4] Silberschatz, Avi and Korth, Henry F. and Sudarshan, S. *Database System Concepts, 6th Edition — Appendix E*. 2010.

[5] Linus Torvalds. *Initial Git commit (Git source repository)*. Git repository commit. 2005.