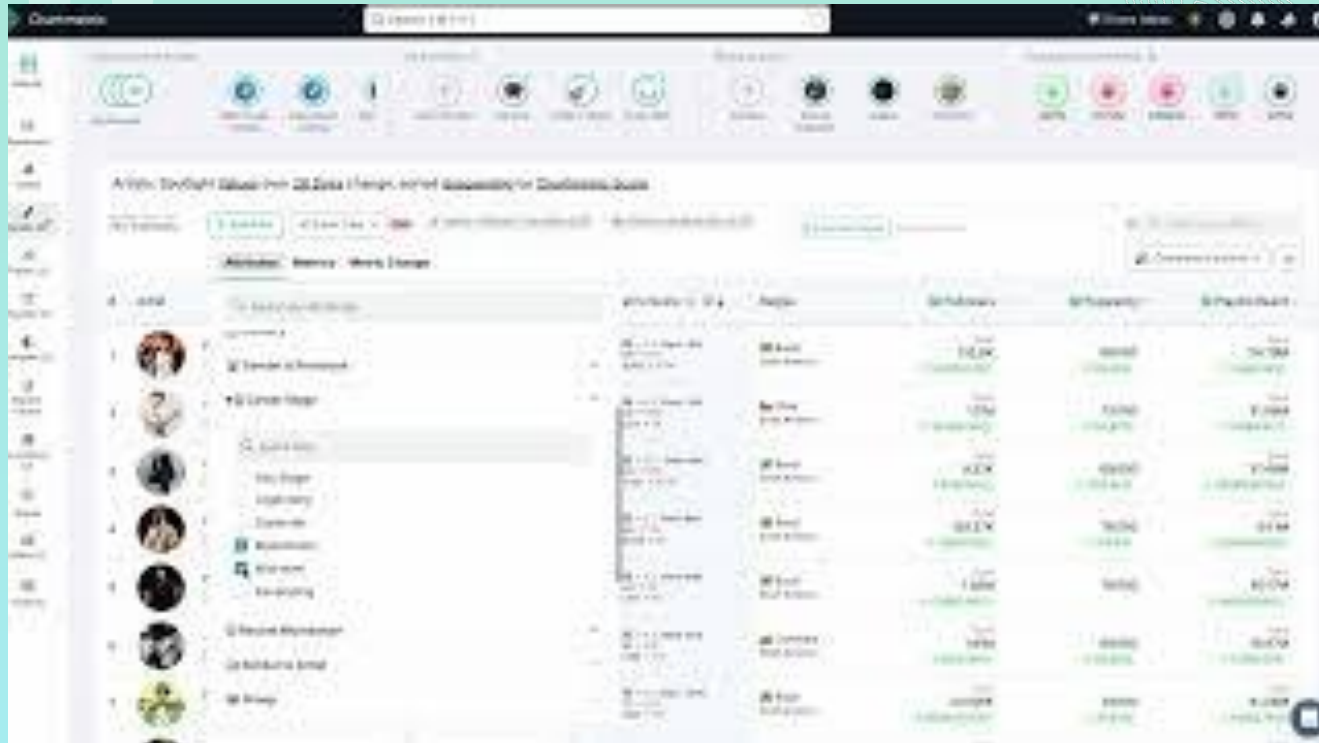




# **Chartmetric & ClickHouse: Data for the Music Industry**

**Peter Gomez**  
**Lead Engineer at Chartmetric**  
**March 19, 2025**

# What we do: Music analytics



[illegible]

# Key data: Streaming platforms, social media, and charts



# Scale and Usage

## Scaling controls

### Service size

ClickHouse Cloud will automatically scale your total memory depending on your needs, use the controls below if you would like to override with precise limits.

[Learn more](#)

Minimum vertical scaling (per node)

64 GiB, 16 vCPU



Maximum vertical scaling (per node)

64 GiB, 16 vCPU



Number of replicas

3 replicas



[Contact support](#) to scale horizontally

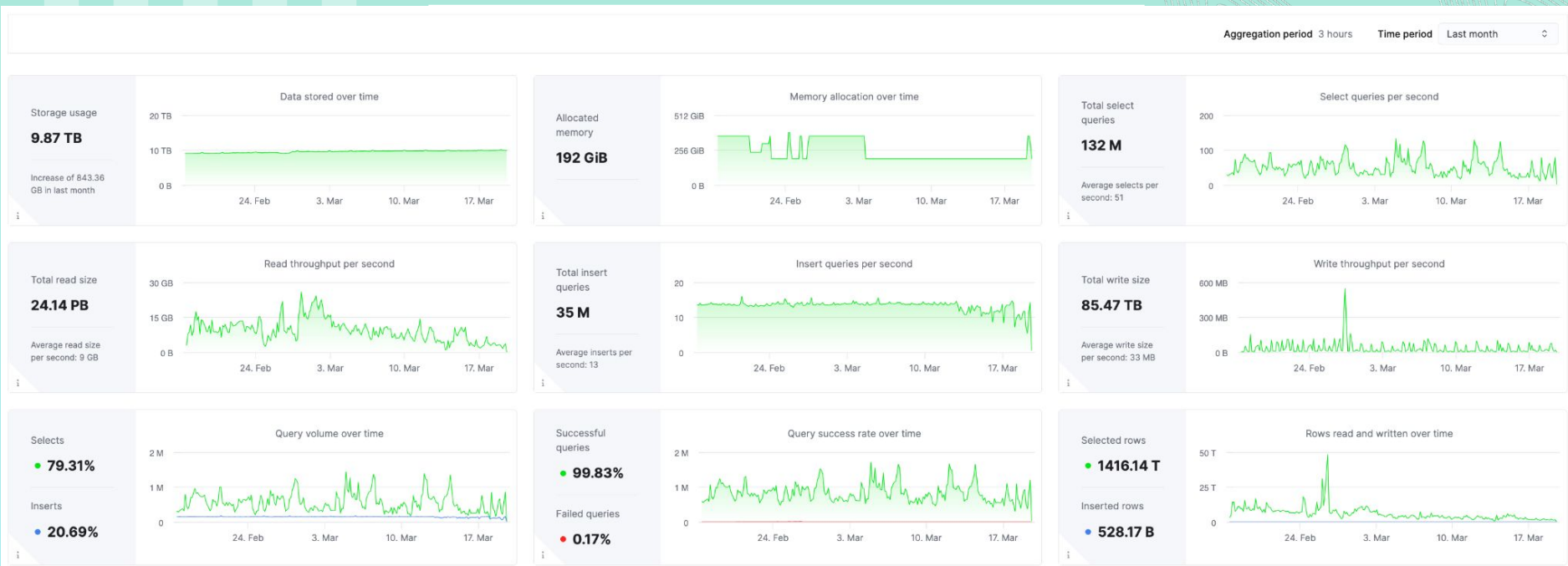
---

Your service will be pinned at 192 GiB, 48 vCPU.

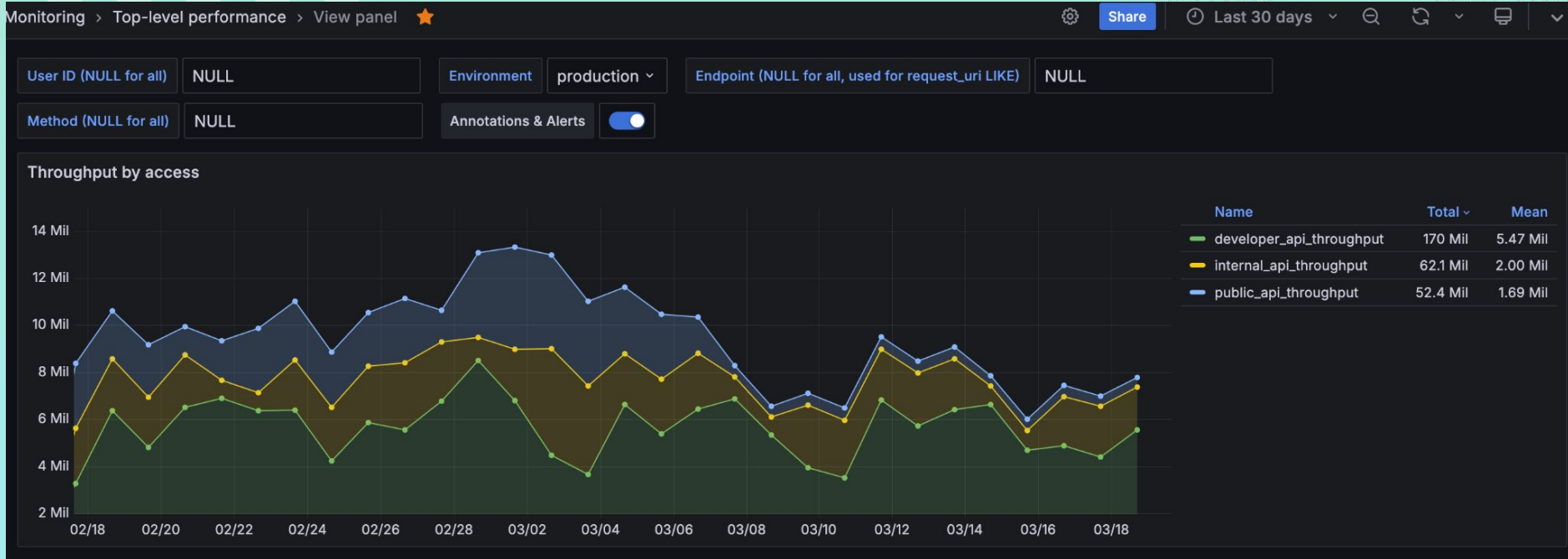




# Scale and Usage



# Scale and Usage



# From Self-hosted to The Cloud

---

```
commit d6f869ea4c23343fbacbcd5705daa44aad542026
```

```
Author: Peter Gomez <fadedlamp42@gmail.com>
```

```
Date: Tue Apr 23 12:08:47 2024 -0700
```

```
initial commit with monitoring and clickhouse code
```

- We began with a single node "cluster" on EC2
  - A **t2.medium** with 30 GB grew into an **m7i.2xlarge** with 8 TB
- This was also the true beginning of our observability journey using Prometheus and Grafana, previously relying on CloudWatch and NewRelic
- We quickly realized the value of ClickHouse Cloud and migrated smoothly
- We also use the ClickHouse OTEL exporter and store all of our logs and metrics there (which we use with the delightful ClickHouse Grafana plugin!)



# Observability with OpenTelemetry

table	compressed	uncompressed	compression_ratio	rows	part_count
otel_logs	270.59 GiB	9.70 TiB	36.7	16.40 billion	363
otel_metrics_sum	121.42 GiB	6.68 TiB	56.36	14.48 billion	327
otel_metrics_gauge	57.99 GiB	4.76 TiB	84.1	9.33 billion	322
otel_traces	30.66 KiB	352.32 KiB	11.49	697.00	2
otel_traces_trace_id_ts	3.08 KiB	5.25 KiB	1.7	131.00	1

- Inspired by last meetup's talks, we now also use the ClickHouse OTEL exporter and store all of our logs and metrics there (which we use with the delightful ClickHouse Grafana plugin!)

# Migrating Time Series Tables from PostgreSQL

---

Our initial success with ClickHouse began with:

- Seamless migration of time series tables due to the amazing ingestion speed
- Created a migration script for SF/RDS to ClickHouse schema conversion

Addressing performance issues:

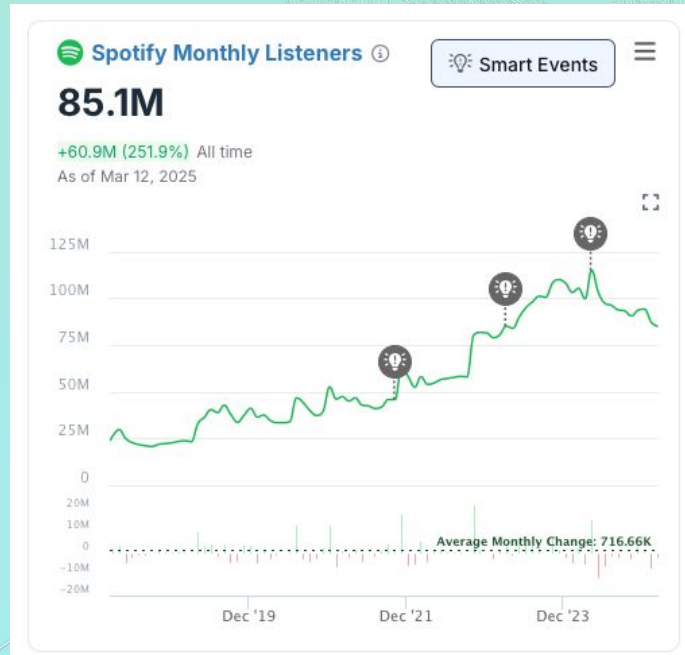
- Initial sync issues due to table ordering based on our API usage
- **Solution:** Created projections based on `max(created_at)` for daily syncs
  - Quickly find the latest data in the table, regardless of the API queries
  - Allows for implementing idempotent, retrievable logic more efficiently

















# Impact of Time Series Table Migrations

Quantifiable benefits:

- Reduced RDS storage from 28TB to 18TB (after migrating ~8 tables)
- A previous query using Snowflake frequently timed out after 1m when users selected the "all" time range (pulling data from 2016)
- With ClickHouse, runtime immediately reduced to **6s** with no caching



# Complex use case: Artist-playlist data

Show: 10 20 50 100					
11 to 20 of 1,512,035 track placements in playlists					
Export					
Playlist	Track	Date Added	Position	Peak Position	Followers ↓
 Songs to Sing in t... (P)  Spotify	Wildest Dreams	Apr 19, 2024 329 Days in List	167/200	136	Total 7.6M ↑ 12.4K(0.2%)
 Top Gaming Tracks (P)  Spotify	I Can Do It With a Broken Heart	Aug 20, 2024 206 Days in List	58/100	26	Total 6.8M ↓ -1K(-0%)
 Pop Up (P)  Spotify	Cruel Summer	Mar 27, 2024 352 Days in List	28/100	1	Total 6.8M ↑ 5.7K(0.1%)
 Pop Up (P)  Spotify	Blank Space (Taylor's Version)	Mar 27, 2024 352 Days in List	76/100	2	Total 6.8M ↑ 5.7K(0.1%)
 Happy Hits! (P)  Spotify	I Forgot That You Existed	Jun 13, 2024 274 Days in List	56/100	3	Total 6.3M ↑ 45.4K(0.7%)
 Happy Hits! (P)  Spotify	Cruel Summer	Jun 13, 2024 274 Days in List	1/100	1	Total 6.3M ↑ 45.4K(0.7%)
 Workout (P)  Spotify	Karma	Oct 8, 2024 157 Days in List	100/100	55	Total 4.8M ↑ 764(0%)

# Complex use case: Artist-playlist data

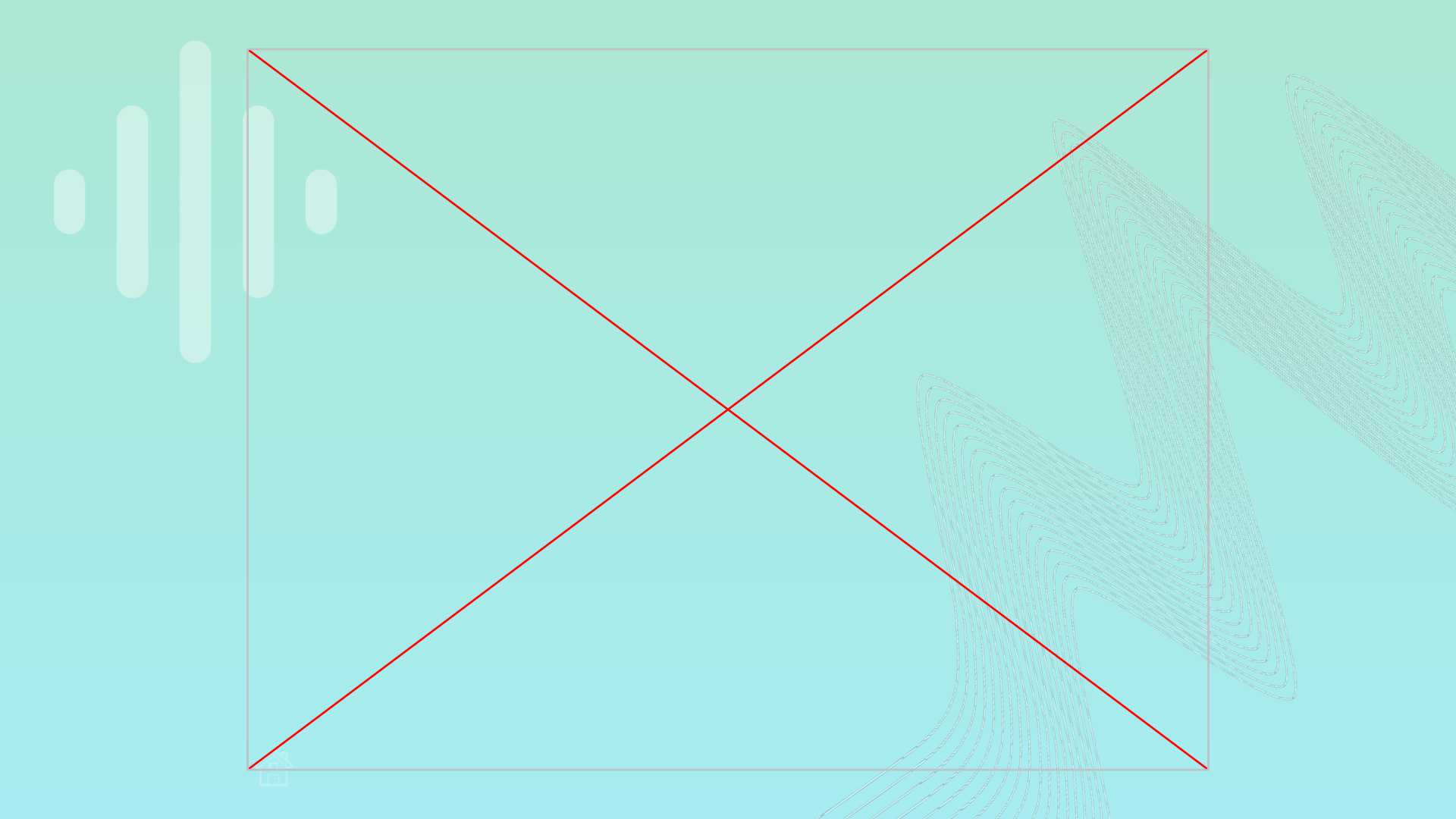
---

## Challenges:

- Complex ingestion pattern
  - INSERT, UPDATE, and DELETE
- Complex querying patterns
  - API can query by artist, track, or album IDs
- Historical data access: 2016 to now
- Multiple filtering options: editorial, radio, algorithmic, etc.
- Users expect hourly data updates, especially on *New Music Friday*
- Massive scale: 30M+ playlists, 2B+ rows, 1.1M daily updates
- Demo in the next slide







# Solution: VersionedCollapsingMergeTree

---

- Biggest challenge were large-scale DELETES with MergeTree
- Solved by using the VersionedCollapsingMergeTree engine
- Created a dedicated table in Snowflake to sync with delete markers
- We thought about creating the entire cache table in ClickHouse from the beginning, but we decided that was too complex of a migration for now

Comparing query performance:

- PostgreSQL: 60+ seconds, Taylor Swift queries took >5 minutes.
- Snowflake: ~20 seconds for top artists.
- **ClickHouse: ~2 seconds**



# Solution: VersionedCollapsingMergeTree

---

- Biggest challenge were large-scale DELETES with MergeTree
- Solved by using the VersionedCollapsingMergeTree engine
- Created a dedicated table in Snowflake to sync with delete markers
- We thought about creating the entire cache table in ClickHouse from the beginning, but we decided that was too complex of a migration for now

Comparing query performance:

- PostgreSQL: 60+ seconds, Taylor Swift queries took >5 minutes.
- Snowflake: ~20 seconds for top artists.
- **ClickHouse: ~2 seconds**



# Solution: VersionedCollapsingMergeTree

```
CREATE TABLE chartmetric_analytics.l_spotify_playlist_cache_vcmt  
(
```

```
    `id` Int64,  
    `cm_track` Int32,  
    `cm_artist` Int32,  
    `ordering` Int32,  
    `spotify` Int32,  
    `spotify_playlist` Int32,  
    `position` Int32,  
    `added_at` Date,  
    `period` Int32,  
    `modified_at` DateTime64(3),  
    `code2` FixedString(2),  
    `followers_latest` Int32,  
    `fdiff_month` Int32,  
    `fdiff_percent_month` Float32,  
    `radio` Bool,  
    `personalized` Bool,  
    `fully_personalized` Bool,  
    `official_curator` Bool,  
    `major_curator` Bool,  
    `peak_position` Int32,  
    `editorial` Bool,  
    `chart` Bool,  
    `this_is` Bool,  
    `new_music_friday` Bool,  
    `brand` Bool,  
    `audiobook` Bool,  
    `indie_curator` Bool,  
    `popular_indie_curator` Bool,  
    `followers_added_at` Int32,  
    `row_active` Int8,
```

```
PROJECTION l_spotify_playlist_cache_vcmt_proj_0 (
```

```
SELECT
```

```
    spotify_playlist,  
    spotify,  
    added_at,  
    period,  
    id,  
    modified_at,  
    row_active
```

```
ORDER BY
```

```
    spotify_playlist,  
    added_at,  
    period,  
    id,  
    modified_at,  
    row_active
```

```
)
```

```
ENGINE = SharedVersionedCollapsingMergeTree(
```

```
    '/clickhouse/tables/{uuid}/{shard}', '{replica}',  
    row_active,  
    modified_at
```

```
)
```

```
ORDER BY (cm_artist, cm_track, spotify, id)
```

```
SETTINGS index_granularity = 8192
```

# Solution: VersionedCollapsingMergeTree

---

```
SELECT *, -1 AS row_active
FROM ANALYTICS.I_spotify_playlist_cache
WHERE id IN (
    SELECT id
    FROM RAW_DATA.I_spotify_playlist_history
    WHERE end_sys_period > (
        SELECT
            MAX(modified_at)
        FROM
            ANALYTICS.I_spotify_playlist_cache
    )
)
UNION ALL
```

```
SELECT id
FROM RAW_DATA.I_spotify_playlist
WHERE modified_at > (
    SELECT
        MAX(modified_at)
    FROM
        ANALYTICS.I_spotify_playlist_cache
)
AND id <= (
    SELECT
        MAX(id)
    FROM
        ANALYTICS.I_spotify_playlist_cache
)
)
ORDER BY id
```





# The Challenge of JOINS: f\_tables

---

- Attempt to create denormalized tables for metadata and metrics
- Memory struggles with large table JOINS in ClickHouse
- Lessons learned:
  - JOIN before ClickHouse wherever possible (Snowflake, PostgreSQL)
  - Limit JOINS to one big table per query when not avoidable
  - Use projections to speed up scans when JOINing two big tables



# Projections: Not quite clustering, not quite indexing

---

- Essential for optimizing queries against central tables where there's no single best ORDER BY
- **Huge** flexibility compared to Snowflake's single clustering key
- Hands off implementation instead of creating, maintaining, and referencing multiple views from API code
- Lack of deletions on projections is a very important caveat
  - RefreshableMaterializedViews are a neat approach to this
- Our adopted projection pattern:
  - Table is ordered by the primary API query column
  - Sync projection for `max(created_at)` to optimize pipelines
  - Extra projections for peculiar query patterns with limited columns

# Key Takeaways

---

- ClickHouse has been **excellent** for time-series data, especially reducing RDS storage costs and improving API performance
- PostgreSQL is **still necessary** for maintaining consistency and complex relationships, and complex relational queries are still most at-home there
- VersionedCollapsingMergeTree and strategic use of Snowflake can handle complex delete operations, like those in our playlist cache
- While large JOINS can be challenging, **projections** and clever use of **materialized views** usually offer a way out
- ClickHouse's projection capabilities make it much more **flexible** than Snowflake's clustering for our specific needs



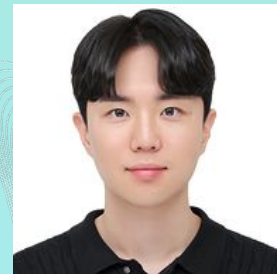
# Thank you!



...and a special thanks to...



**Umang**  
Lead Data Infra Engineer



**Hyosik**  
Senior Software Engineer

...for helping put this presentation together!

An abstract graphic consisting of numerous thin, white, wavy lines that originate from the top-left corner and flow downwards and to the right, creating a sense of movement and depth. The lines are set against a solid teal background.

**Q&A**