Artifact

Adding Caching to the recommendationAlgo Micro-Service

by Radoslav Radev

Advanced Software Engineering Semester 6

13/04/2024

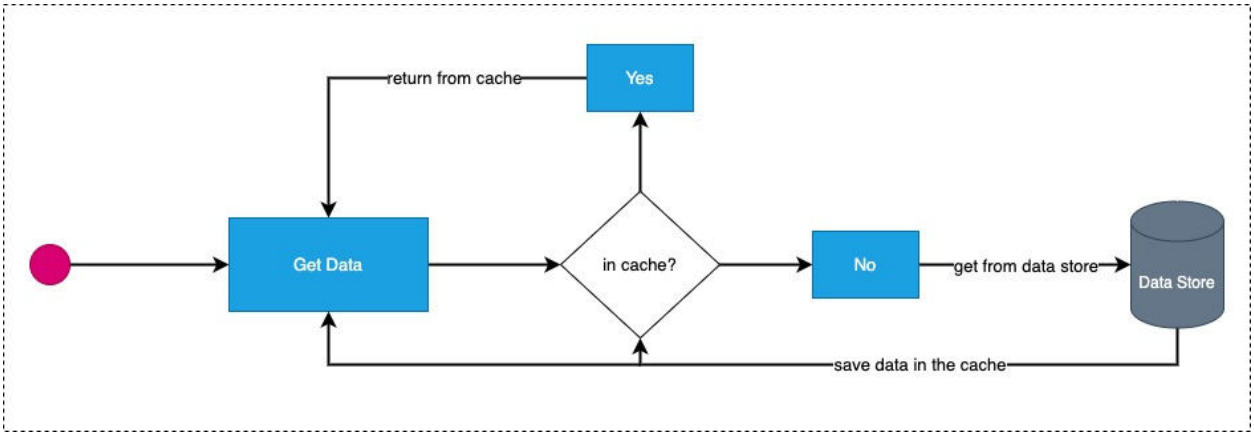
ver. 1.0

Introduction:

I have implemented a Cache-Aside pattern using Redis for the `recommendationAlgo` microservice. The caching strategy focuses on storing computed results rather than raw data, optimizing response times for user recommendations. The data items and their respective caching intervals are as follows:

- User Favorite Categories: 24 hours
- Popular Videos: 24 hours
- Similar Users (those who liked similar videos): 20 minutes
- User's Liked Videos: 20 minutes

By caching these computed datasets, we ensure that the recommendation algorithm recalculates them only after their cache expires. This approach fine-tunes user recommendations approximately every 20 minutes, rather than with every query. We are considering extending the cache lifespan to further reduce load on the system.



Benchmark Results:

The introduction of caching has added a performance increase of approximately 15% with the exception of GetAlgoRecommendationsForUser as evidenced by the following benchmark results:

```
AMD Ryzen 7 6800H with Radeon Graphics, 1 CPU, 16 logical and 8 physical cores
.NET SDK 8.0.202
  [Host]     : .NET 8.0.3 (8.0.324.11423), X64 RyuJIT AVX2
  DefaultJob : .NET 8.0.3 (8.0.324.11423), X64 RyuJIT AVX2


| Method                                        | Mean       | Error    | StdDev   |
|---------------------------------------------- |-----------:|---------:|---------:|
| GetUserPreferredCategories_New                |   387.2 us |  7.54 us |  7.74 us |
| GetUserPreferredCategories_Old                |   443.7 us |  8.81 us | 10.14 us |
| GetPopularVideosInGeneral_New                 |   381.6 us |  2.43 us |  1.90 us |
| GetPopularVideosInGeneral_Old                 |   463.3 us |  7.04 us |  6.58 us |
| GetPopularVideosWithExcludedWatchedForUser_New |   780.6 us |  9.32 us |  7.78 us |
| GetPopularVideosWithExcludedWatchedForUser_Old |   899.1 us | 10.72 us |  9.50 us |
| GetAlgoRecommendationsForUser_New             | 2,839.1 us | 36.18 us | 32.08 us |
| GetAlgoRecommendationsForUser_Old             | 1,836.4 us | 28.32 us | 26.49 us |

// * Hints *
Outliers
  RecommendationRepositoryBenchmark.GetUserPreferredCategories_Old: Default                -> 1 outlier  was  removed (480.21 us)
  RecommendationRepositoryBenchmark.GetPopularVideosInGeneral_New: Default                 -> 3 outliers were removed (394.32 us..397.25 us)
  RecommendationRepositoryBenchmark.GetPopularVideosWithExcludedWatchedForUser_New: Default -> 2 outliers were removed (827.40 us, 827.51 us)
  RecommendationRepositoryBenchmark.GetPopularVideosWithExcludedWatchedForUser_Old: Default -> 1 outlier  was  removed (938.24 us)
  RecommendationRepositoryBenchmark.GetAlgoRecommendationsForUser_New: Default             -> 2 outliers were removed (2.96 ms, 2.97 ms)

// * Legends *
  Mean   : Arithmetic mean of all measurements
  Error  : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  1 us   : 1 Microsecond (0.000001 sec)
```

Method	Mean	Error	StdDev
GetUserPreferredCategories_New	387.2 µs	7.54 µs	7.74 µs
GetUserPreferredCategories_Old	443.7 µs	8.81 µs	10.14 µs
GetPopularVideosInGeneral_New	381.6 µs	2.43 µs	1.90 µs
GetPopularVideosInGeneral_Old	463.3 µs	7.04 µs	6.58 µs
GetPopularVideosWithExcludedWatchedForUser_New	780.6 µs	9.32 µs	7.78 µs

GetPopularVideosWithExcludedWatchedForUser_Old	899.1 µs	10.72 µs	9.50 µs
GetAlgoRecommendationsForUser_New	2,839.1 µs	36.18 µs	32.08 µs
GetAlgoRecommendationsForUser_Old	1,836.4 µs	28.32 µs	26.49 µs

Legends:

Mean : Arithmetic mean of all measurements

Error : Half of 99.9% confidence interval

StdDev : Standard deviation of all measurements

1 us : 1 Microsecond (0.000001 sec)

Outliers:

RecommendationRepositoryBenchmark.GetUserPreferredCategories_Old: Default -> 1 outlier was removed (480.21 us)

RecommendationRepositoryBenchmark.GetPopularVideosInGeneral_New: Default -> 3 outliers were removed (394.32 us..397.25 us)

RecommendationRepositoryBenchmark.GetPopularVideosWithExcludedWatchedForUser_New : Default -> 2 outliers were removed (827.40 us, 827.51 us)

RecommendationRepositoryBenchmark.GetPopularVideosWithExcludedWatchedForUser_Old: Default -> 1 outlier was removed (938.24 us)

RecommendationRepositoryBenchmark.GetAlgoRecommendationsForUser_New: Default -> 2 outliers were removed (2.96 ms, 2.97 ms)

Analysis and Future Work:

The GetAlgoRecommendationsForUser method initially performed all calculations within the database, resulting in a single request. However, with caching, this has changed to two separate calls to Redis, followed by a simpler database query, totaling three requests. The additional requests for "similarUsers" and "likedVideos" have inadvertently slowed performance.

To address this,I can instead add a 20-minute cache for algorithm recommended videos per user, and on request filter by excluding already watched videos. This modification should reduce database load while maintaining or improving performance.

```
public async Task<List<Guid>> GetAlgoRecommendedVideos(Guid accId, int topN)
{
    // Cache keys
    var likedVideosCacheKey = $"LikedVideos-{accId}";
    var similarUsersCacheKey = $"SimilarUsers-{accId}";

    // Attempt to fetch liked videos from cache
    List<Guid> likedVideos = await GetFromCache<List<Guid>>(likedVideosCacheKey);
    if (likedVideos == null)
    {
        likedVideos = await dbContext.WatchHistories // DbSet<WatchHistory>
            .Where(w => w.UserId == accId && w.Liked == VideoLikeEnum.Liked) // IQueryable<WatchHistory>
            .Select(w => w.VideoId).ToListAsync(); // Task<List<...>>

        await SetCache(likedVideosCacheKey, likedVideos, expiration: TimeSpan.FromMinutes(20));
    }


    // Attempt to fetch similar users from cache
    List<Guid> similarUsers = await GetFromCache<List<Guid>>(similarUsersCacheKey);
    if (similarUsers == null)
    {
        similarUsers = await dbContext.WatchHistories // DbSet<WatchHistory>
            .Where(w => likedVideos.Contains(w.VideoId) && w.UserId != accId && w.Liked == VideoLikeEnum.Liked) // IQueryable<WatchHistory>
            .Select(w => w.UserId).Distinct().ToListAsync(); // Task<List<...>>

        await SetCache(similarUsersCacheKey, similarUsers, expiration: TimeSpan.FromMinutes(20));
    }




    // Define a query that calculates the popularity score for each video directly in the database
    var query // IQueryable<(VideoId,PopularityScore...)> = dbContext.WatchHistories // DbSet<WatchHistory>
        .GroupJoin(dbContext.VideoStats, outerKeySelector: wh => wh.VideoId, innerKeySelector: vs => vs.VideoId,
            resultSelector: (wh, vs) => new { WatchHistory = wh, VideoStats = vs.FirstOrDefault() }) // IQueryable<(WatchHistory,VideoStats)>
        .GroupBy(w => w.WatchHistory.VideoId).Select(g => new
        {
            VideoId = g.Key,
            PopularityScore =
                (g.Count(w => w.WatchHistory.Liked == VideoLikeEnum.Liked) /
                (decimal)(g.Count(w => w.WatchHistory.Liked == VideoLikeEnum.Liked) +
                          g.Count(w => w.WatchHistory.Liked == VideoLikeEnum.Disliked))) +
                (g.Sum(w => w.WatchHistory.WatchedTime) / g.Max(w => w.VideoStats.VideoLength)),
            LikeSimilarityScore = dbContext.WatchHistories // DbSet<WatchHistory>
                .Where(w => similarUsers.Contains(w.UserId) && w.Liked == VideoLikeEnum.Liked &&
                            !likedVideos.Contains(w.VideoId)).Count(),
            Category = g.Max(w => w.VideoStats.Category)
        })
```

Conclusion:

The integration of caching into the recommendationAlgo microservice has demonstrated promising results, improving efficiency and response times for user recommendations. Ongoing

optimizations and adjustments will focus on enhancing performance further, particularly for the GetAlgoRecommendationsForUser method.