Strategies for Optimal Performance: Advanced Techniques in Entity Framework for High-Volume Query Optimization

CS4991 Capstone Report, 2024

Ahyush Kaul
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
Sdx6cq@virginia.edu

ABSTRACT

Navigating billions of database records presents significant challenges and costs, even with refined search parameters tailored to meet imposed runtime limitations. Implementing sophisticated optimization techniques appeared as a viable solution to address the challenges associated with querying larger amounts of records during my time at CoStar Group. To optimize the process, I employed sophisticated techniques, utilizing C# .NET's Entity Framework to strategically limit the retrieval of records from the database. Complementing this, I implemented a continuous cache emptying technique, facilitating the seamless integration of new records. The optimization strategies netted an efficiency increase of 180x, drastically improving the query runtime. While optimizing runtime is crucial for efficiency, advancing query optimizations to include the pagination of data can help make query searches more accurate for CoStar's data needs.

1. INTRODUCTION

Databases are crucial in modern information management systems, as they offer a structured and organized repository to store and retrieve data efficiently. The significance of databases lies in their ability to facilitate the quick retrieval of data, allowing an organization to access/manipulate information rapidly. Compared to a traditional file based system, a database uses queries to sort through records to efficiently store and return data. The rapid access databases provide is pivotal for meeting the demands of fast-paced data-driven businesses.

CoStar collects billions of real estate records(data points) and houses all information in a database. When developers require information, they can write a query to access specific records in the database. However, with such a vast quantity of data entries, narrowing down the exact data required can pose a challenge. Even after creating targeted queries, accessing the data can take longer than allowed by front-end services. This issue arose when an in-house API route ran for more than 30 minutes, causing a time-out error on the front-end for an internal tool. The time-out limitation allowed for five seconds before returning an error, and solving this issue was crucial to fix a bug that prevented hundreds of developers from accessing crucial business information.

2. RELATED WORKS

According to Balliauw (2023), one of the main components for slow EF (Entity Framework) queries is returning too many records. When creating queries to access records, we must consider the total number of available records to retrieve. From there, we must select certain attributes to filter from in order to limit the number of records selected. There is a caveat however. When writing queries that require access to another table in the database, we risk cartesian explosion,

which happens when more data than necessary is loaded, causing worse performance (Nguyen, 2023). The command AsSplitQuery() helps to solve this issue, as it will separate the larger query into smaller components. These smaller portions are then run separately to retrieve specific entries from the database. This solution dramatically improves runtime and reduces the risk of selecting the same record multiple times.

When a user interface needs to show results for a query from a larger dataset, pagination may be required (Microsoft, 2023). Pagination occurs when data is received in pages instead of one bulk return. To implement pagination the query must have a Take() command. This restricts the amount of records that the query will return, which can improve performance since a smaller amount of data is retrieved from the database. The solution, while intuitive, is limited in scope, as it requires a Filter() command to sort the data in a manner where the most effective values are returned. This poses an additional issue: how to determine which records are most valuable.

3. PROJECT DESIGN

This project required multiple stages which necessitated learning about query optimization techniques, front-end service limitations, and API route constraints. Additionally, other developers and I had to conduct vigorous testing to ensure its success.

3.1 Background

The query in question was part of an API route to receive a collection of data processed by researchers at CoStar. This query had been written months prior to my arrival and was a high priority task. Due to the inefficiency of the query, not all CoStar employees could see the results of processed data on the internal tool to which the data was returned.

3.2 API Route Debugging

The initial step involved tracing the issue on the front-end aspect to the API route call. To accomplish this, I examined the failing console statements associated with the error on the website. After tracing the error to the API call in the repository associated with the front-end code, I transitioned to the repository housing the API routes. Examining the specific API route connected to the error revealed a complex EF query. This query resulted in cartesian explosion as it included multiple sub tables associated with the main table. The collection of this data resulted in an inefficient query that could not process the vast amount of data quickly.

3.3 EF Query Improvements

Solving this issue required research into EF query structure. The nature of the problem was multi-faceted; including issues with pagination, data overload, and poor filtering. To improve the query I broke each issue up, focusing first on pagination. Utilizing a Take() command at the end of the query helped limit the data collected and returned, shortening runtime.

This then presented an issue with filtering the data to gather the most relevant records. Incorporating a filter by date in the query allowed me to gather the most recent records preserving the relevance of the data. This also fit with corporate business decisions to showcase the most relevant data points from researchers.

Another issue was data persistence in the cache. As data was being received from the database and stored in the API route to be returned, it took up space in the cache. With the

quantity of data being searched and gathered by the query, cache space became scarce. When data matched the query, it had a higher likelihood of becoming a cache miss and evict. This process, while quick, aggregated to take more time than allotted, contributing to poor performance. To solve this, we added the commands AsNoTracking() and AsSplitQuery() after the Take() command. These methods resulted in data not being cached and separating larger queries to become more manageable, respectively.

4. RESULTS

After all optimization were added, testing had to be conducted to ensure the API route no longer timed out. Since the API route has automated search parameters, we automated testing to run the API route through Postman and a Swagger page. Loading the API route through its Swagger page allowed us to see the automated search parameters, output format, and actual output. This is a crucial part in determining the accuracy and efficiency of the route.

To verify the information returned through the API was accurate, we ran the query directly in the database using PostgreSQL. The API route query and SQL query returned the same information, validating the response expected. Additionally, the API route returned much faster compared to its non-optimized form. Through optimizations the route improved its runtime by 180,000% and produced a memory optimization of 20%.

The drastic improvement in runtime was associated with the limitation of returned data, lack of caching required, and separation of queries. The memory optimization was a non-intentional effect of the caching optimization, as data from the query was able to be placed in the cache without needing to evict other data. This meant that evicted data did not need to be stored elsewhere, such as on the disk.

5. CONCLUSION

This project was significant to CoStar as it expands its business and allowed all employees to have efficient access to crucial business information. The query optimizations utilized will ensure quick and easy data retrieval as the database grows. With the expansion of their business, CoStar's database will also grow and having this scalable code in place will help maintain the operationality of an internal tool used by all employees.

The query had been problematic for CoStar but was backlogged in tasks due to heavy demand from CoStar consumers. By working on the optimization of this query, I was able to tackle a task that otherwise would not have been completed for months. This project increased my knowledge of database interactions and how data is collected through querying. It also taught me a great deal about database management and research techniques towards real-world niche problems that I can apply in future works with CoStar.

6. FUTURE WORK

Next steps for the query optimizations would be to apply them to other API search routes. CoStar has various API's they develop, with most of them querying the same larger database. Adding optimization like AsNoTracking and AsSplitQuery can help drastically improve runtime across all API search routes.

Additionally, it would be beneficial to research better methods for pagination. Currently we are limited in the scope of data the API route can access with the Take command. Finding a better alternative to remove the necessity of filtering may

reduce runtime even more. Optimizing the routes further can help reduce time spent waiting for values to be returned, creating a better user experience.

REFERENCES

Balliauw, M. (2023, March 2). *Optimizing entity framework core database queries with Dynamic Program Analysis: The .net tools blog*. The JetBrains Blog. https://blog.jetbrains.com/dotnet/2023/03/02/optimizing-entity-framework-core-database-queries-with-dynamic-program-analysis/#:~:text=Slow%20queries%20in%20Entity%20Framework,defaults%20to%20500%20ms)%3A

Nguyen, F. (2023, February 27). *Understanding the cartesian explosion issue in EF Core*. Medium. https://medium.com/knowledge-pills/understanding-the-cartesian-explosion-issue-in-ef-core-1e6091483a43

Microsoft. (2023, January 12). *Efficient querying - EF core*. EF Core | Microsoft Learn. https://learn.microsoft.com/en-us/ef/core/performance/efficient-querying