Optimizing APIs: Using Precomputation to Reduce Complexity and Latency

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Jared Nguyen

Spring, 2022

Technical Project Team Members

Jared Nguyen

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

MC Forelle, Department of Engineering and Society

Daniel Graham, Computer Science

Optimizing APIs: Using Precomputation to Reduce Complexity and Latency

CS4991 Capstone Report, 2023

Jared Nguyen
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
jtn2km@virginia.edu

ABSTRACT

A company that provides a service storing user's files faced latency issues with an API that returns a given user's file usage. The issues stemmed from the current implementation of the API essentially "counting" each individual file per customer. To remedy the issue, I worked with a team of software engineers to optimize the API by precomputing these values into a database to reduce the latency to a database read operation. This process involved identifying and modifying key components of the Java Spring backend of the product and interacting with existing and new AWS infrastructure. The P99 latency of the API was reduced from 2 seconds to 40 milliseconds and removed an estimated 800,000 daily file usage aggregation operations from a stressed datastore. Implementing the project in production will involve adding existing user usage into the database and testing for edge cases resulting from data inconsistency and scale.

1. INTRODUCTION

Have you ever navigated to a website, only to be met with a loading spinner? After waiting a couple of seconds, you likely grew frustrated and navigated away. That was what the experience was like for the growing userbase of the user file storage service, a total number reaching the tens of millions. Worst-case latency was estimated to soon exceed recommended load times of 0-4 seconds (Baker, 2022).

As a result, my internship project focused on optimizing the API which returned customer file usage. The current implementation of the API computed a customer's file usage by aggregating individual file sizes. This operation was executed as a query to a data store containing user file metadata. It was facing intense load, as it was used by a multitude of other internal and external services.

Attempts to reduce latency and stress on the datastore already existed in the form of a caching mechanism, where user usage values existed for several minutes in a cache, before being considered stale and requiring a re-computation of values. However, in the worst-case scenario, users could potentially update or delete a file and a change to their usage values would not be visible until the cached values expire after several minutes — a significant detriment to user experience.

The proposed technical project solution focused on removing the dependence on the stressed data store. Its main goal was to precompute the user usage values instead of relying on "on-the-fly" aggregation. This would be done by maintaining a running total of usage values per user stored in a new external database, then examining each

individual file upload or delete to increment or decrement their usage values. Then, the API could simply read the precomputed value from the database.

2. RELATED WORK

A major notable work that inspired the project was an internal company document detailing an investigation into why the data store was performing poorly, resulting in latency issues. The document revealed how the data store used indexing and sharding. Indexing is a technique to speed up queries without having to search every element in the datastore. The index key used an improperly used data type resulting in slow lookups. Sharding is a technique of partitioning data across servers, which can increase response times. However, historically, the data store was one of the largest ever in size, resulting in excessive number of shards, which brings drawbacks, like data inconsistencies and hot spotting. As a result, this document inspired my intern project aimed at removing the dependency on this data store.

Precomputation has a wide range of uses and applications from compilers, machine learning, and databases. As a result, there is no single seminal usage of the technique, though it is broadly used in industry. For example, a blog post from the Uber Data/ML team describes the chronology of the engineering effort to develop a routing algorithm for their application (Uber, 2015). The algorithm needed to be fast, while handling real-time traffic. The article describes how routing algorithms represent real-world intersections and roads as a graph with nodes and edges, respectively. Edges have a weight to them, usually the time to travel across the road, along with other heuristic information. Finding an ETA for a route consists of summing up the weights of the edges to the destination. The article describes using contraction hierarchies, a method to speed this process up, which relies on precomputing "shortcut" paths and storing them as an additional edge, so this does not have to be done in real-time.

3. PROJECT DESIGN

The following subsections describe the system architecture of the current system and the project solution, as well as a discussion of the challenges faced during the transition.

3.1 Current System Architecture

The default flow of the file usage API consisted of a query to the datastore containing user file metadata. The datastore was considered the source of truth within the scope of the project. It aggregates the sizes of all the user's files and returns that result. Attempts to reduce the number of these expensive computations existed in the form of a caching mechanism, where the cache periodically stored the computed usage values. First, the file usage API would read from the cache, if the timestamp (representing the last time the usage value was computed) exceeded a certain number of minutes, the API would proceed to query the datastore as normal. As shown in Figure 1, the newly computed value would be reinserted into the cache, overwriting the stale usage value.

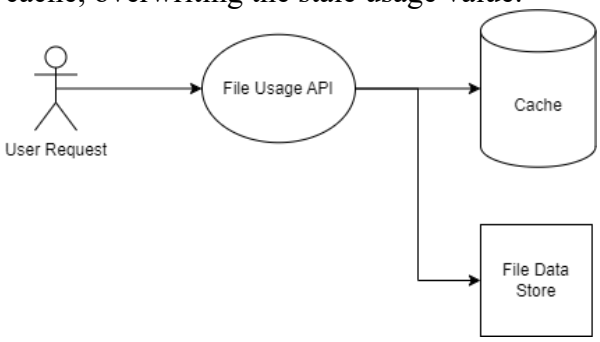

Figure 1: Current Architecture with Caching

3.2 System Architecture of Project Solution

The project solution would remove the caching mechanism and the dependence on the file data store completely. Instead, a pre-existing data stream that provides information about individual file updates was utilized. The file update data stream represented a user uploading, deleting, or modifying a file, along

with the size of the resultant file. These properties were used to decide whether to increment or decrement the running total of a user's file usage, which was stored in a new database containing precomputed values. As seen in Figure 2, the Precomputed Usage Manager represents the application code that performs the precomputation as described above.

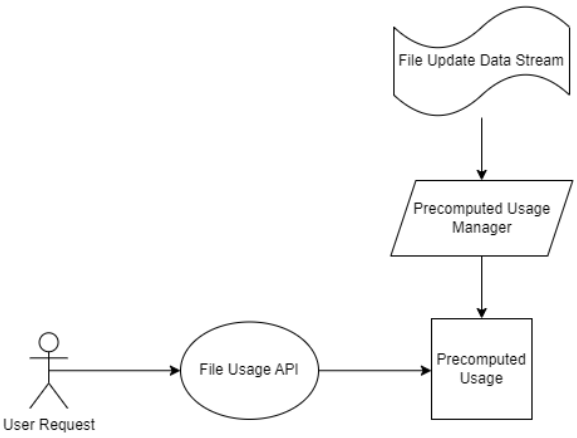

Figure 2: Project Architecture with Precomputation

3.3 Challenges

Many challenges occurred due to the nature of the architecture as a distributed system. Instances of application code run across multiple hosts as a means of horizontal scaling (Splunk, 2022). As a result, a user's usage value may be precomputed concurrently on different machines, resulting in a race condition (Vagdevi, 2021). To address this issue, version numbers were used to determine if a race condition had occurred. A version number was assigned to each usage value that increments upon modification. By checking the current version number was unchanged before writing to the database, it can be determined a race condition occurred, in which the entire process can be retried.

Another challenge resulted from integrating into the existing application and the size of the userbase. Individual file updates were presented as a difference between "old" and "new" files. For example, uploading a file was represented as the old file having a default values, such as having a file size of zero with no name, to the new file having the expected properties of the uploaded file. As a result, it was difficult to directly determine the intended operation of the file update and heuristics were used instead. However, due to the large number of possible states a file could be in, in addition to the large userbase increasing the likelihood of an edge case occurring, it was difficult to create a definitive implementation that manages every possible file update state.

4. RESULTS

A significant outcome of my project was removing the dependence of the user file usage API on the stressed data store. This resulted in a significant reduction of daily file usage aggregation operations by an estimated 800,000. By reducing the number of calls to the stressed datastore, other remaining components dependent on it will generally benefit as it has a larger bandwidth to support these components. The implementation also removes the necessity of the caching mechanism, which will eventually be phased out as the project implementation moves to production. This directly resolves the major user experience issue of stale file usage values being displayed for several minutes, as value will now be instantaneously updated. Furthermore, it provides the additional benefit of reducing the P99 latency from 2 seconds to a database read latency, which is around 40 milliseconds. Many of the user mobile, web, and desktop applications rely on the file usage API, as well as other internal APIs; these will widely benefit from reduced latency times. Overall, this project served as a part of a larger effort to break up the monolithic backend.

5. CONCLUSION

The file usage API is a vital component of the system architecture and has an extensive history of maintenance throughout the lifetime of the legacy service provided by the company. Due to a wide dependence on this

API throughout the system, my project was proposed to resolve latency and user experience issues. By using precomputation, the P99 latency of the API was reduced from 2 seconds to 40 milliseconds and updates to a user's usage became instantaneous. It is important to remember that, while these benefits are significant, the previous implementation of the API served adequately throughout the lifetime of the application. Only due to the increasing scale of the userbase did these issues emerge. The system architecture of an application should be constantly evolving as it responds to changing requirements.

6. FUTURE WORK

To ensure the project implementation is ready for release to the broader user base, extensive work needs to be done. Since a new database was created to maintain the users' file usage, it does not contain existing usage values for users. There exist two approaches to populate the database with this data. One is database backfilling, which is the process of filling data from the past in a new system (Chen, 2022). This can be done all at once with a script or another external process that transforms the historical data into the newer database, however, it can be costly, time-consuming, and may require downtime as there are millions of users. The second approach is to fill this data in as users request usage data from the API. Filling in usage value upon request will gradually update users' usage values into the database, but still depends on the previous implementation of the file usage API. As a result, there continues to be many questions to be answered to fully release these changes.

REFERENCES

Baker, K. (2022, April 7). *11 Website Page Load Time Statistics You Need [+ How to Increase Conversion Rate]*. Hubspot.com; HubSpot. https://blog.hubspot.com/marketing/page-load-time-conversion-rates

Chen, B. (2022, November 9). The Data Engineer's Guide To Backfilling Data. Retrieved April 18, 2023, from Monte Carlo Data website: https://www.montecarlodata.com/blog-backfilling-data-guide/

ETA Phone Home: How Uber Engineers an Efficient Route. (2015, November 3). Uber Blog. https://www.uber.com/blog/engineering-routing-engine/

Splunk. (2022). Splunk; Splunk. https://www.splunk.com/en_us/data-insider/what-are-distributed-systems.html

Vagdevi K. (2021, April 21). *Race Condition and Deadlock | CloudxLab Blog*. CloudxLab Blog. https://cloudxlab.com/blog/race-condition-and-deadlock/