






No Caching antipattern

06/05/2017 • 8 minutes to read • Contributors     

In this article

[Problem description](#)

[How to fix the problem](#)

[Considerations](#)

[How to detect the problem](#)

[Example diagnosis](#)

[Related resources](#)

In a cloud application that handles many concurrent requests, repeatedly fetching the same data can reduce performance and scalability.


Problem description

When data is not cached, it can cause a number of undesirable behaviors, including:

- Repeatedly fetching the same information from a resource that is expensive to access, in terms of I/O overhead or latency.
- Repeatedly constructing the same objects or data structures for multiple requests.
- Making excessive calls to a remote service that has a service quota and throttles clients past a certain limit.

In turn, these problems can lead to poor response times, increased contention in the data store, and poor scalability.

The following example uses Entity Framework to connect to a database. Every client request results in a call to the database, even if multiple requests are fetching exactly the same data. The cost of repeated requests, in terms of I/O overhead and data access charges, can accumulate quickly.

C#	 Copy
<pre>public class PersonRepository : IPersonRepository { public async Task<Person> GetAsync(int id) { using (var context = new AdventureWorksContext()) { return await context.People .Where(p => p.Id == id) .FirstOrDefaultAsync() .ConfigureAwait(false); } } }</pre>	

You can find the complete sample [here](#).

This antipattern typically occurs because:

- Not using a cache is simpler to implement, and it works fine under low loads. Caching makes the code more complicated.
- The benefits and drawbacks of using a cache are not clearly understood.
- There is concern about the overhead of maintaining the accuracy and freshness of cached data.

- An application was migrated from an on-premises system, where network latency was not an issue, and the system ran on expensive high-performance hardware, so caching wasn't considered in the original design.
- Developers aren't aware that caching is a possibility in a given scenario. For example, developers may not think of using ETags when implementing a web API.

How to fix the problem

The most popular caching strategy is the *on-demand* or *cache-aside* strategy.

- On read, the application tries to read the data from the cache. If the data isn't in the cache, the application retrieves it from the data source and adds it to the cache.
- On write, the application writes the change directly to the data source and removes the old value from the cache. It will be retrieved and added to the cache the next time it is required.

This approach is suitable for data that changes frequently. Here is the previous example updated to use the [Cache-Aside](#) pattern.

C# 

```
public class CachedPersonRepository : IPersonRepository
{
    private readonly PersonRepository _innerRepository;

    public CachedPersonRepository(PersonRepository innerRepository)
    {
        _innerRepository = innerRepository;
    }

    public async Task<Person> GetAsync(int id)
    {
        return await CacheService.GetAsync<Person>("p:" + id, () =>
_innerRepository.GetAsync(id)).ConfigureAwait(false);
    }
}

public class CacheService
{
    private static ConnectionMultiplexer _connection;

    public static async Task<T> GetAsync<T>(string key, Func<Task<T>> loadCache, double expirationTimeInMinutes)
    {
        IDatabase cache = Connection.GetDatabase();
        T value = await GetAsync<T>(cache, key).ConfigureAwait(false);
        if (value == null)
        {
            // Value was not found in the cache. Call the lambda to get the value from the
            database.
            value = await loadCache().ConfigureAwait(false);
            if (value != null)
            {
                // Add the value to the cache.
                await SetAsync(cache, key, value,
expirationTimeInMinutes).ConfigureAwait(false);
            }
        }
        return value;
    }
}
```

Notice that the `GetAsync` method now calls the `CacheService` class, rather than calling the database directly. The `CacheService` class first tries to get the item from Azure Redis Cache. If the value isn't found in Redis Cache, the

CacheService invokes a lambda function that was passed to it by the caller. The lambda function is responsible for fetching the data from the database. This implementation decouples the repository from the particular caching solution, and decouples the CacheService from the database.

Considerations

- If the cache is unavailable, perhaps because of a transient failure, don't return an error to the client. Instead, fetch the data from the original data source. However, be aware that while the cache is being recovered, the original data store could be swamped with requests, resulting in timeouts and failed connections. (After all, this is one of the motivations for using a cache in the first place.) Use a technique such as the [Circuit Breaker pattern](#) to avoid overwhelming the data source.
- Applications that cache nonstatic data should be designed to support eventual consistency.
- For web APIs, you can support client-side caching by including a Cache-Control header in request and response messages, and using ETags to identify versions of objects. For more information, see [API implementation](#).
- You don't have to cache entire entities. If most of an entity is static but only a small piece changes frequently, cache the static elements and retrieve the dynamic elements from the data source. This approach can help to reduce the volume of I/O being performed against the data source.
- In some cases, if volatile data is short-lived, it can be useful to cache it. For example, consider a device that continually sends status updates. It might make sense to cache this information as it arrives, and not write it to a persistent store at all.
- To prevent data from becoming stale, many caching solutions support configurable expiration periods, so that data is automatically removed from the cache after a specified interval. You may need to tune the expiration time for your scenario. Data that is highly static can stay in the cache for longer periods than volatile data that may become stale quickly.
- If the caching solution doesn't provide built-in expiration, you may need to implement a background process that occasionally sweeps the cache, to prevent it from growing without limits.
- Besides caching data from an external data source, you can use caching to save the results of complex computations. Before you do that, however, instrument the application to determine whether the application is really CPU bound.
- It might be useful to prime the cache when the application starts. Populate the cache with the data that is most likely to be used.
- Always include instrumentation that detects cache hits and cache misses. Use this information to tune caching policies, such what data to cache, and how long to hold data in the cache before it expires.
- If the lack of caching is a bottleneck, then adding caching may increase the volume of requests so much that the web front end becomes overloaded. Clients may start to receive HTTP 503 (Service Unavailable) errors. These are an indication that you should scale out the front end.

How to detect the problem

You can perform the following steps to help identify whether lack of caching is causing performance problems:

1. Review the application design. Take an inventory of all the data stores that the application uses. For each, determine whether the application is using a cache. If possible, determine how frequently the data changes. Good initial candidates for caching include data that changes slowly, and static reference data that is read frequently.

2. Instrument the application and monitor the live system to find out how frequently the application retrieves data or calculates information.
3. Profile the application in a test environment to capture low-level metrics about the overhead associated with data access operations or other frequently performed calculations.
4. Perform load testing in a test environment to identify how the system responds under a normal workload and under heavy load. Load testing should simulate the pattern of data access observed in the production environment using realistic workloads.
5. Examine the data access statistics for the underlying data stores and review how often the same data requests are repeated.

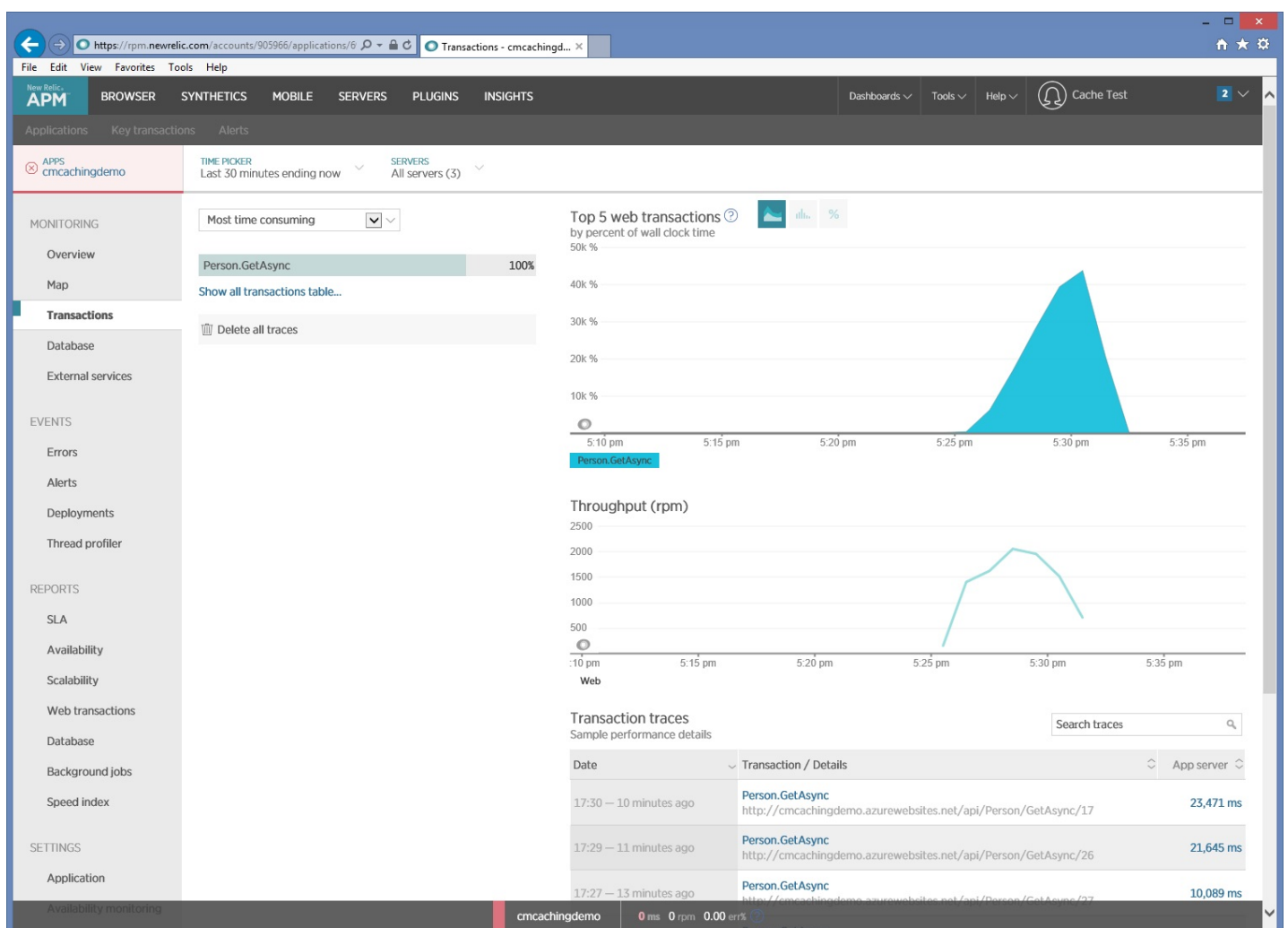
Example diagnosis

The following sections apply these steps to the sample application described earlier.

Instrument the application and monitor the live system

Instrument the application and monitor it to get information about the specific requests that users make while the application is in production.

The following image shows monitoring data captured by [New Relic](#) during a load test. In this case, the only HTTP GET operation performed is `Person/GetAsync`. But in a live production environment, knowing the relative frequency that each request is performed can give you insight into which resources should be cached.

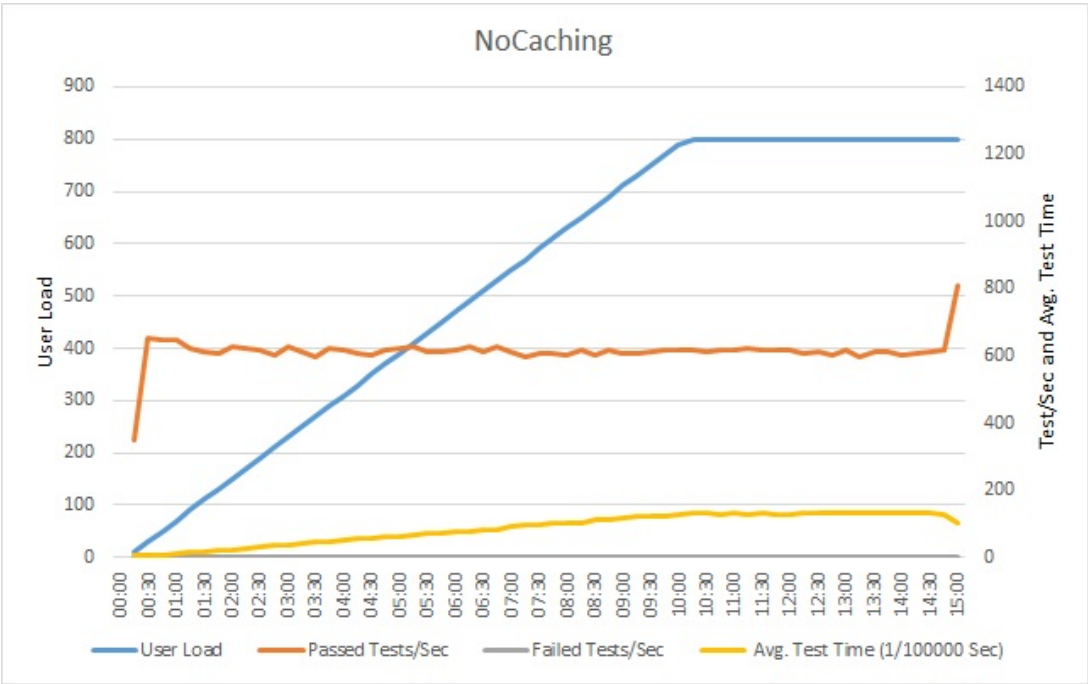


If you need a deeper analysis, you can use a profiler to capture low-level performance data in a test environment (not the production system). Look at metrics such as I/O request rates, memory usage, and CPU utilization. These metrics

may show a large number of requests to a data store or service, or repeated processing that performs the same calculation.

Load test the application

The following graph shows the results of load testing the sample application. The load test simulates a step load of up to 800 users performing a typical series of operations.



The number of successful tests performed each second reaches a plateau, and additional requests are slowed as a result. The average test time steadily increases with the workload. The response time levels off once the user load peaks.

Examine data access statistics

Data access statistics and other information provided by a data store can give useful information, such as which queries are repeated most frequently. For example, in Microsoft SQL Server, the `sys.dm_exec_query_stats` management view has statistical information for recently executed queries. The text for each query is available in the `sys.dm_exec_query_plan` view. You can use a tool such as SQL Server Management Studio to run the following SQL query and determine how frequently queries are performed.

SQL	Copy
<pre>SELECT UseCounts, Text, Query_Plan FROM sys.dm_exec_cached_plans CROSS APPLY sys.dm_exec_sql_text(plan_handle) CROSS APPLY sys.dm_exec_query_plan(plan_handle)</pre>	

The `UseCount` column in the results indicates how frequently each query is run. The following image shows that the third query was run more than 250,000 times, significantly more than any other query.

```

SELECT UseCounts, Text, Query_Plan
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)

```

100 %

Results Messages

	UseCounts	Text	Query_Plan
1	1	SELECT UseCounts, Text, Query_Plan FROM sys.dm_...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
2	1	select is_federation_member from sys.databases where ...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
3	256049	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
4	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
5	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
6	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
7	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
8	3	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
9	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
10	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
11	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
12	3	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
13	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
14	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
15	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
16	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
17	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
18	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
19	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
20	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
21	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
22	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
23	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
24	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...

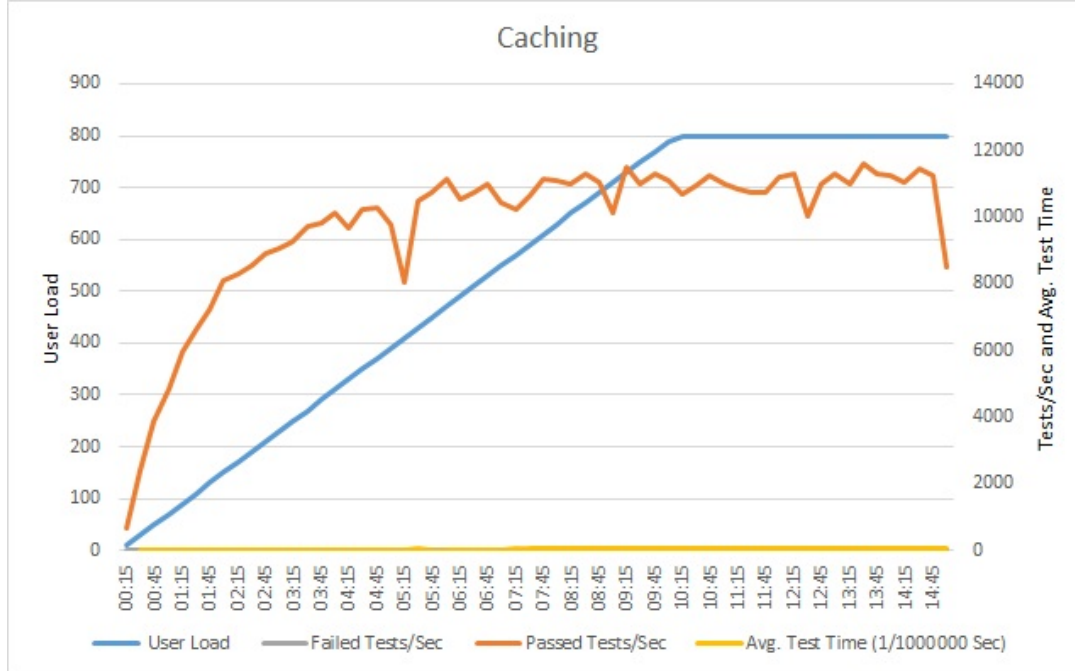
Here is the SQL query that is causing so many database requests:

SQL	Copy
<pre> (@p__linq__0 int)SELECT TOP (2) [Extent1].[BusinessEntityId] AS [BusinessEntityId], [Extent1].[FirstName] AS [FirstName], [Extent1].[LastName] AS [LastName] FROM [Person].[Person] AS [Extent1] WHERE [Extent1].[BusinessEntityId] = @p__linq__0 </pre>	

This is the query that Entity Framework generates in `GetByIdAsync` method shown earlier.

Implement the solution and verify the result

After you incorporate a cache, repeat the load tests and compare the results to the earlier load tests without a cache. Here are the load test results after adding a cache to the sample application.



The volume of successful tests still reaches a plateau, but at a higher user load. The request rate at this load is significantly higher than earlier. Average test time still increases with load, but the maximum response time is 0.05 ms, compared with 1 ms earlier—a 20× improvement.

Related resources

- [API implementation best practices](#)
- [Cache-Aside pattern](#)
- [Caching best practices](#)
- [Circuit Breaker pattern](#)