




Extraneous Fetching antipattern

06/05/2017 • 10 minutes to read • Contributors    

In this article

[Problem description](#)

[How to fix the problem](#)

[Considerations](#)

[How to detect the problem](#)

[Example diagnosis](#)

[Implement the solution and verify the result](#)

[Related resources](#)

Retrieving more data than needed for a business operation can result in unnecessary I/O overhead and reduce responsiveness.

Problem description

This antipattern can occur if the application tries to minimize I/O requests by retrieving all of the data that it *might* need. This is often a result of overcompensating for the [Chatty I/O](#) antipattern. For example, an application might fetch the details for every product in a database. But the user may need just a subset of the details (some may not be relevant to customers), and probably doesn't need to see *all* of the products at once. Even if the user is browsing the entire catalog, it would make sense to paginate the results—showing 20 at a time, for example.

Another source of this problem is following poor programming or design practices. For example, the following code uses Entity Framework to fetch the complete details for every product. Then it filters the results to return only a subset of the fields, discarding the rest. You can find the complete sample [here](#).

```
public async Task<IHttpActionResult> GetAllFieldsAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Execute the query. This happens at the database.
        var products = await context.Products.ToListAsync();

        // Project fields from the query results. This happens in application memory.
        var result = products.Select(p => new ProductInfo { Id = p.ProductId, Name = p.Name });
        return Ok(result);
    }
}
```

In the next example, the application retrieves data to perform an aggregation that could be done by the database instead. The application calculates total sales by getting every record for all orders sold, and then computing the sum over those records. You can find the complete sample [here](#).

```
public async Task<IHttpActionResult> AggregateOnClientAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Fetch all order totals from the database.
        var orderAmounts = await context.SalesOrderHeaders.Select(soh => soh.TotalDue).ToListAsync()
```

```

sync();

    // Sum the order totals in memory.
    var total = orderAmounts.Sum();
    return Ok(total);
}
}

```

The next example shows a subtle problem caused by the way Entity Framework uses LINQ to Entities.

```

var query = from p in context.Products.AsEnumerable()
            where p.SellStartDate < DateTime.Now.AddDays(-7) // AddDays cannot be mapped by
LINQ to Entities
            select ...;

List<Product> products = query.ToList();

```

The application is trying to find products with a `SellStartDate` more than a week old. In most cases, LINQ to Entities would translate a `where` clause to a SQL statement that is executed by the database. In this case, however, LINQ to Entities cannot map the `AddDays` method to SQL. Instead, every row from the `Product` table is returned, and the results are filtered in memory.

The call to `AsEnumerable` is a hint that there is a problem. This method converts the results to an `IEnumerable` interface. Although `IEnumerable` supports filtering, the filtering is done on the *client* side, not the database. By default, LINQ to Entities uses `IQueryable`, which passes the responsibility for filtering to the data source.

How to fix the problem

Avoid fetching large volumes of data that may quickly become outdated or might be discarded, and only fetch the data needed for the operation being performed.

Instead of getting every column from a table and then filtering them, select the columns that you need from the database.

```

public async Task<IHttpActionResult> GetRequiredFieldsAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Project fields as part of the query itself
        var result = await context.Products
            .Select(p => new ProductInfo { Id = p.ProductId, Name = p.Name })
            .ToListAsync();
        return Ok(result);
    }
}

```

Similarly, perform aggregation in the database and not in application memory.

```

public async Task<IHttpActionResult> AggregateOnDatabaseAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Sum the order totals as part of the database query.
        var total = await context.SalesOrderHeaders.SumAsync(soh => soh.TotalDue);
    }
}

```

```
        return Ok(total);  
    }  
}
```

When using Entity Framework, ensure that LINQ queries are resolved using the `IQueryable` interface and not `IEnumerable`. You may need to adjust the query to use only functions that can be mapped to the data source. The earlier example can be refactored to remove the `AddDays` method from the query, allowing filtering to be done by the database.

```
DateTime dateSince = DateTime.Now.AddDays(-7); // AddDays has been factored out.  
var query = from p in context.Products  
            where p.SellStartDate < dateSince // This criterion can be passed to the database  
            by LINQ to Entities  
            select ...;  
  
List<Product> products = query.ToList();
```

Considerations

- In some cases, you can improve performance by partitioning data horizontally. If different operations access different attributes of the data, horizontal partitioning may reduce contention. Often, most operations are run against a small subset of the data, so spreading this load may improve performance. See [Data partitioning](#).
- For operations that have to support unbounded queries, implement pagination and only fetch a limited number of entities at a time. For example, if a customer is browsing a product catalog, you can show one page of results at a time.
- When possible, take advantage of features built into the data store. For example, SQL databases typically provide aggregate functions.
- If you're using a data store that doesn't support a particular function, such as aggregation, you could store the calculated result elsewhere, updating the value as records are added or updated, so the application doesn't have to recalculate the value each time it's needed.
- If you see that requests are retrieving a large number of fields, examine the source code to determine whether all of these fields are necessary. Sometimes these requests are the result of poorly designed `SELECT *` query.
- Similarly, requests that retrieve a large number of entities may be sign that the application is not filtering data correctly. Verify that all of these entities are needed. Use database-side filtering if possible, for example, by using `WHERE` clauses in SQL.
- Offloading processing to the database is not always the best option. Only use this strategy when the database is designed or optimized to do so. Most database systems are highly optimized for certain functions, but are not designed to act as general-purpose application engines. For more information, see the [Busy Database antipattern](#).

How to detect the problem

Symptoms of extraneous fetching include high latency and low throughput. If the data is retrieved from a data store, increased contention is also probable. End users are likely to report extended response times or failures caused by services timing out. These failures could return HTTP 500 (Internal Server) errors or HTTP 503 (Service Unavailable) errors. Examine the event logs for the web server, which likely contain more detailed information about the causes and circumstances of the errors.

The symptoms of this antipattern and some of the telemetry obtained might be very similar to those of the [Monolithic Persistence antipattern](#).

You can perform the following steps to help identify the cause:

1. Identify slow workloads or transactions by performing load-testing, process monitoring, or other methods of capturing instrumentation data.
2. Observe any behavioral patterns exhibited by the system. Are there particular limits in terms of transactions per second or volume of users?
3. Correlate the instances of slow workloads with behavioral patterns.
4. Identify the data stores being used. For each data source, run lower-level telemetry to observe the behavior of operations.
5. Identify any slow-running queries that reference these data sources.
6. Perform a resource-specific analysis of the slow-running queries and ascertain how the data is used and consumed.

Look for any of these symptoms:

- Frequent, large I/O requests made to the same resource or data store.
- Contention in a shared resource or data store.
- An operation that frequently receives large volumes of data over the network.
- Applications and services spending significant time waiting for I/O to complete.

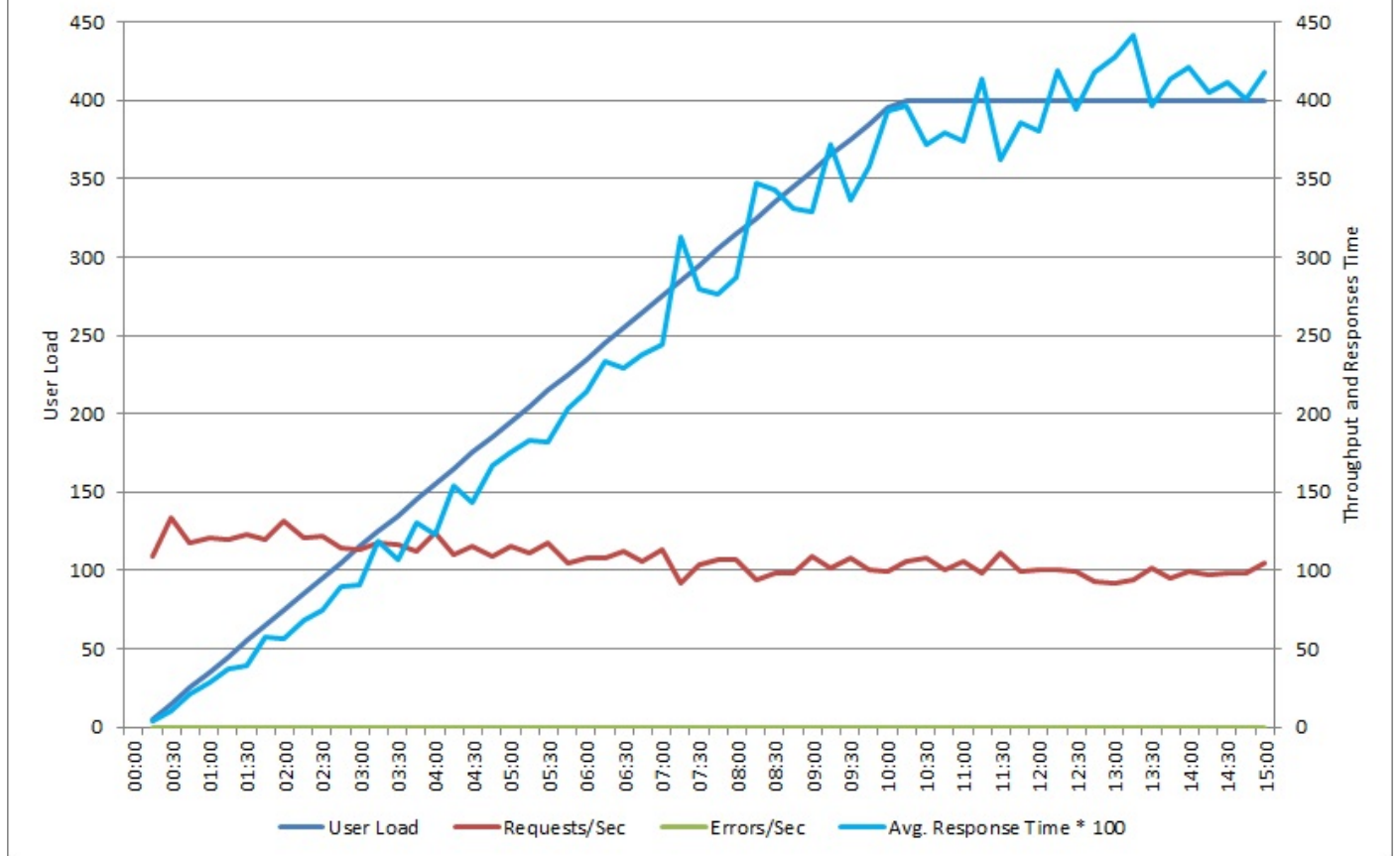
Example diagnosis

The following sections apply these steps to the previous examples.

Identify slow workloads

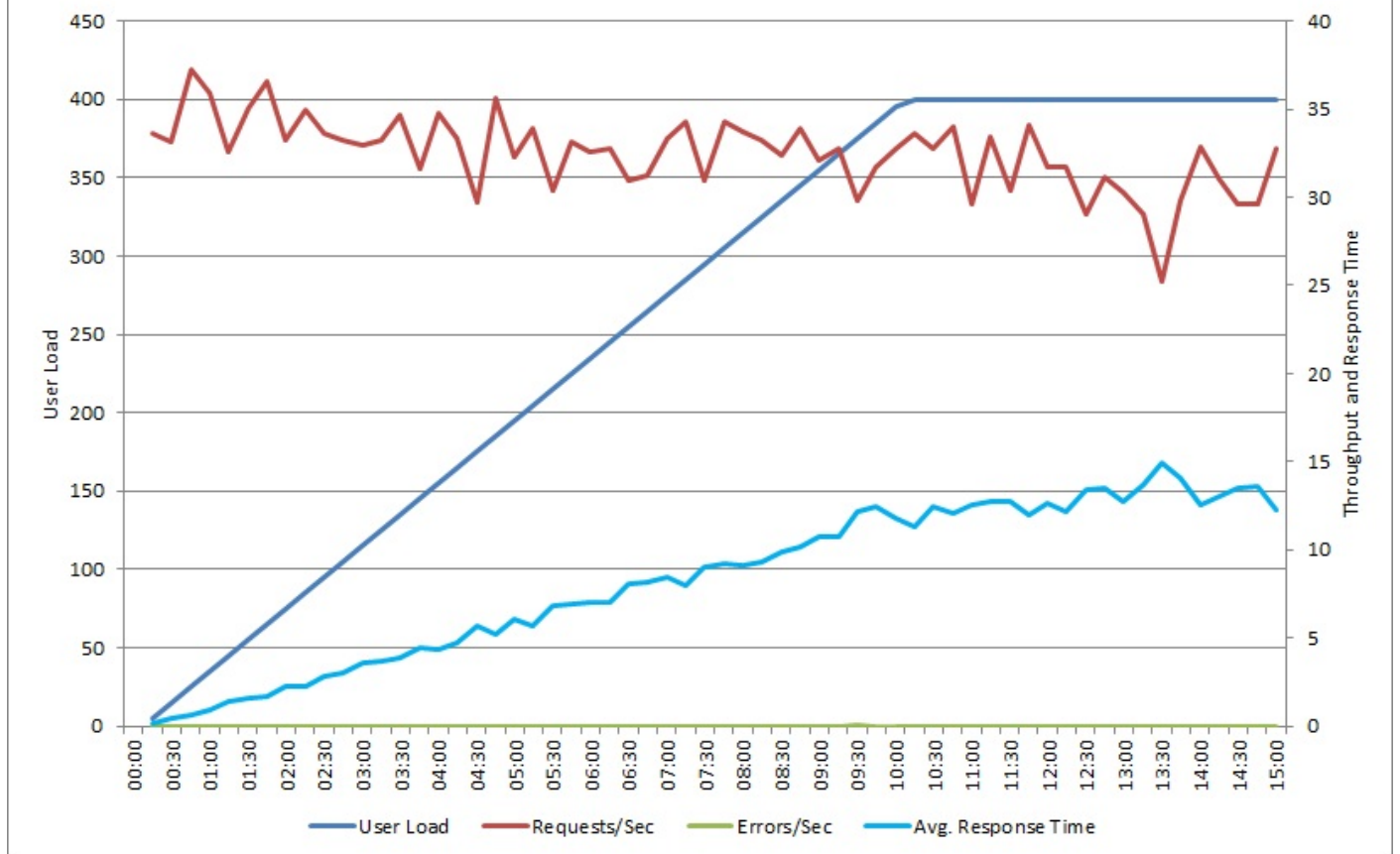
This graph shows performance results from a load test that simulated up to 400 concurrent users running the `GetAllFieldsAsync` method shown earlier. Throughput diminishes slowly as the load increases. Average response time goes up as the workload increases.

Retrieving All Fields



A load test for the `AggregateOnClientAsync` operation shows a similar pattern. The volume of requests is reasonably stable. The average response time increases with the workload, although more slowly than the previous graph.

Aggregating On Client



Correlate slow workloads with behavioral patterns

Any correlation between regular periods of high usage and slowing performance can indicate areas of concern. Closely examine the performance profile of functionality that is suspected to be slow running, to determine whether it matches

the load testing performed earlier.

Load test the same functionality using step-based user loads, to find the point where performance drops significantly or fails completely. If that point falls within the bounds of your expected real-world usage, examine how the functionality is implemented.

A slow operation is not necessarily a problem, if it is not being performed when the system is under stress, is not time critical, and does not negatively affect the performance of other important operations. For example, generating monthly operational statistics might be a long-running operation, but it can probably be performed as a batch process and run as a low-priority job. On the other hand, customers querying the product catalog is a critical business operation. Focus on the telemetry generated by these critical operations to see how the performance varies during periods of high usage.

Identify data sources in slow workloads

If you suspect that a service is performing poorly because of the way it retrieves data, investigate how the application interacts with the repositories it uses. Monitor the live system to see which sources are accessed during periods of poor performance.

For each data source, instrument the system to capture the following:

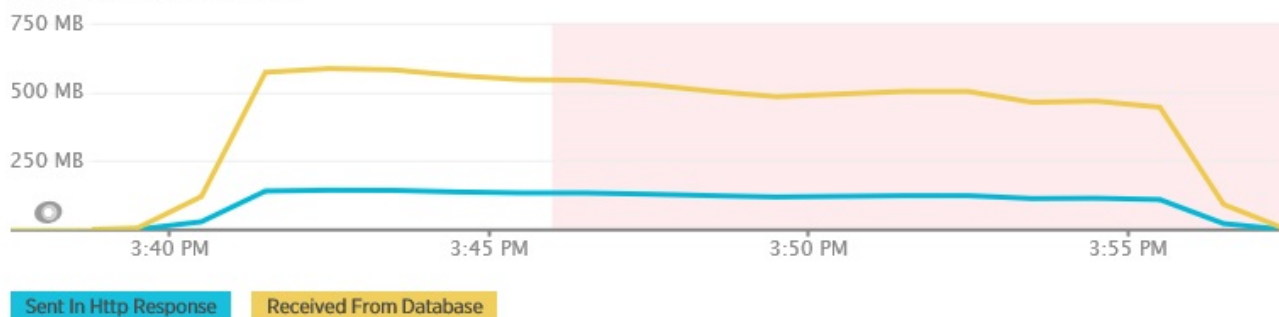
- The frequency that each data store is accessed.
- The volume of data entering and exiting the data store.
- The timing of these operations, especially the latency of requests.
- The nature and rate of any errors that occur while accessing each data store under typical load.

Compare this information against the volume of data being returned by the application to the client. Track the ratio of the volume of data returned by the data store against the volume of data returned to the client. If there is any large disparity, investigate to determine whether the application is fetching data that it doesn't need.

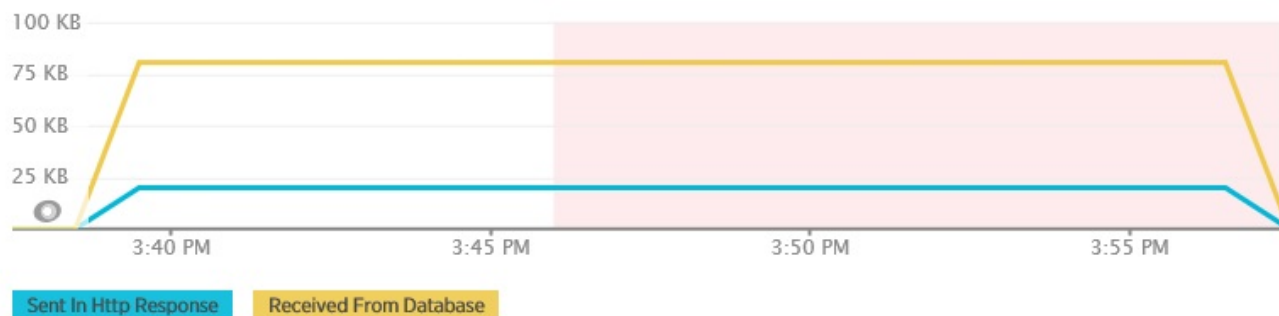
You may be able to capture this data by observing the live system and tracing the lifecycle of each user request, or you can model a series of synthetic workloads and run them against a test system.

The following graphs show telemetry captured using [New Relic APM](#) during a load test of the `GetAllFieldsAsync` method. Note the difference between the volumes of data received from the database and the corresponding HTTP responses.

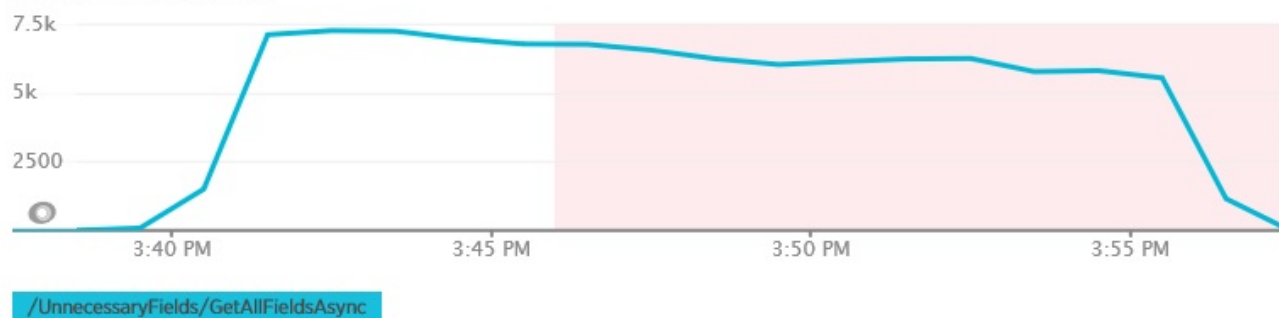
Total Bytes per Minute



Average Bytes per Transaction



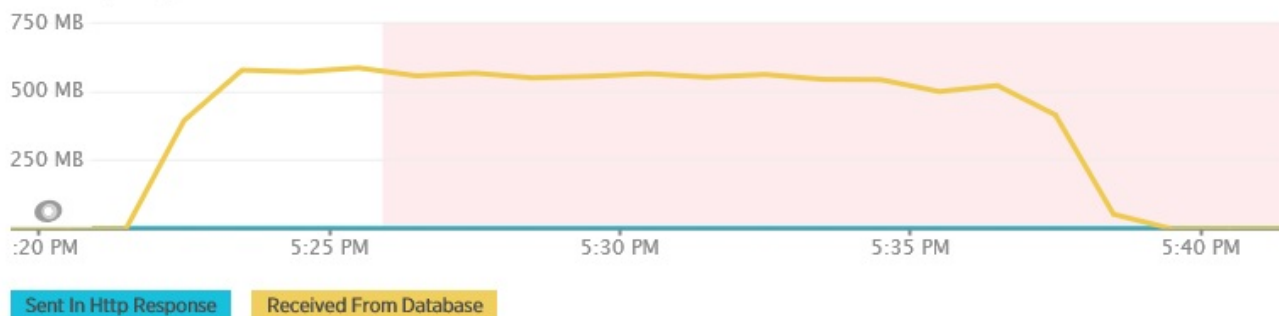
Requests per Minute



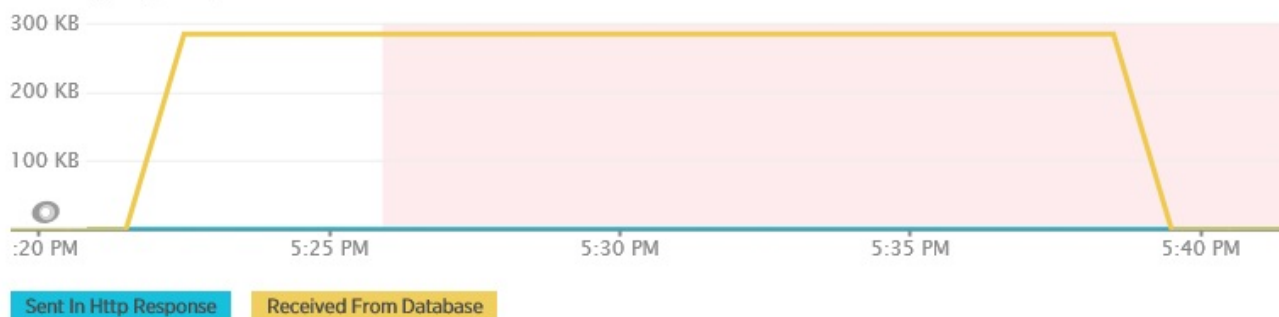
For each request, the database returned 80,503 bytes, but the response to the client only contained 19,855 bytes, about 25% of the size of the database response. The size of the data returned to the client can vary depending on the format. For this load test, the client requested JSON data. Separate testing using XML (not shown) had a response size of 35,655 bytes, or 44% of the size of the database response.

The load test for the `AggregateOnClientAsync` method shows more extreme results. In this case, each test performed a query that retrieved over 280 Kb of data from the database, but the JSON response was a mere 14 bytes. The wide disparity is because the method calculates an aggregated result from a large volume of data.

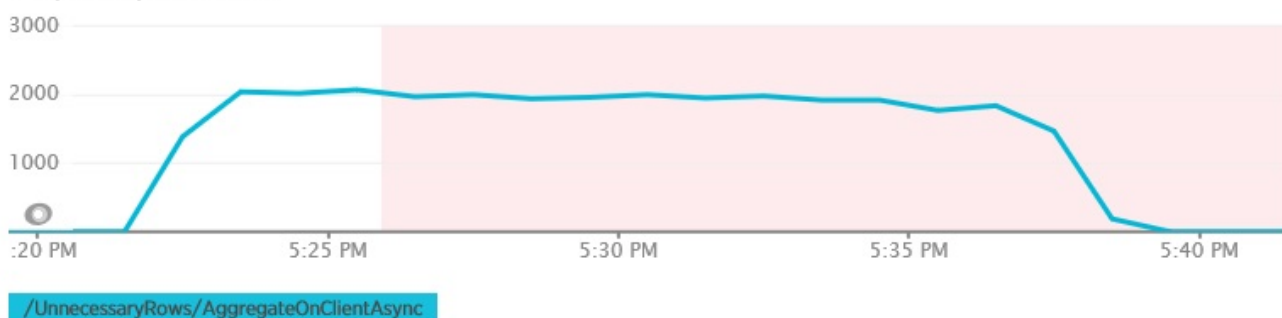
Total Bytes per Minute



Average Bytes per Transaction



Requests per Minute



Identify and analyze slow queries

Look for database queries that consume the most resources and take the most time to execute. You can add instrumentation to find the start and completion times for many database operations. Many data stores also provide in-depth information on how queries are performed and optimized. For example, the Query Performance pane in the Azure SQL Database management portal lets you select a query and view detailed runtime performance information. Here is the query generated by the `GetAllFieldsAsync` operation:


```

SELECT
    [Extent1].[ProductID] AS [ProductID],
    [Extent1].[Name] AS [Name],
    [Extent1].[ProductNumber] AS [ProductNumber],
    [Extent1].[MakeFlag] AS [MakeFlag],
    [Extent1].[FinishedGoodsFlag] AS [FinishedGoodsFlag],
    [Extent1].[Color] AS [Color],
    [Extent1].[SafetyStockLevel] AS [SafetyStockLevel],
    [Extent1].[ReorderPoint] AS [ReorderPoint],
    [Extent1].[StandardCost] AS [StandardCost],
    [Extent1].[ListPrice] AS [ListPrice],
    [Extent1].[Size] AS [Size],
    [Extent1].[SizeUnitMeasureCode] AS [SizeUnitMeasureCode],
    [Extent1].[WeightUnitMeasureCode] AS [WeightUnitMeasureCode],
    [Extent1].[Weight] AS [Weight],

```

[Query Plan Details](#)
[Query Plan](#)

Resource Use

Resource	Total / sec	Total	Last Run	Minimum	Maximum
Duration (ms)	117	63450	40	0	3867
CPU (ms)	1	752	0	0	3
Logical Reads	25	13872	16	16	16
Physical Reads	0	0	0	0	0
Logical Writes	0	0	0	0	0

Plan Information

Run Count	867
Last Run Time	03/03/2015 15:32:25
Plan Generation Count	1
Time Plan Cached	03/03/2015 15:26:32

Advanced Information

Plan Handle	0x0600E0004D00CD02A0EDD87D5500000001000000000000000000000000
SQL Handle	0x020000004D00CD02B85696A75DA7BEBF79E79259988F30C300000000
Query Hash	0x05ACF4F9056ACADC
Query Plan Hash	0x0DA11AA10A268A7B

Implement the solution and verify the result

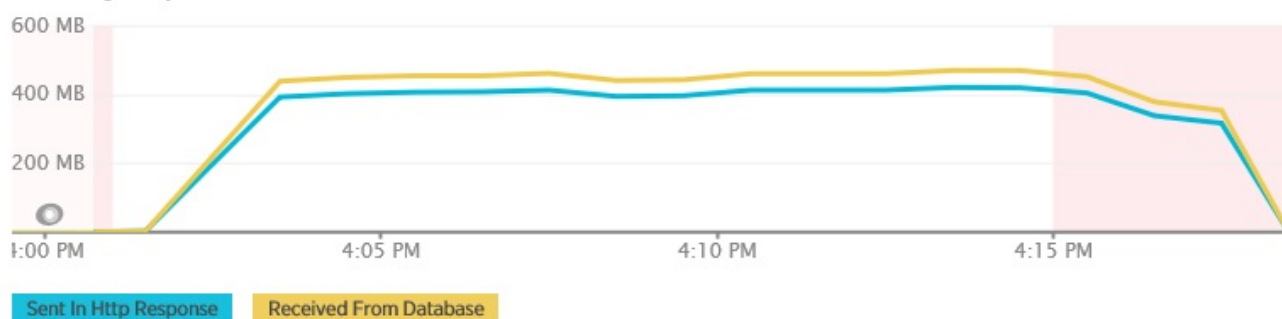
After changing the GetRequiredFieldsAsync method to use a SELECT statement on the database side, load testing showed the following results.

Retrieving Required Fields

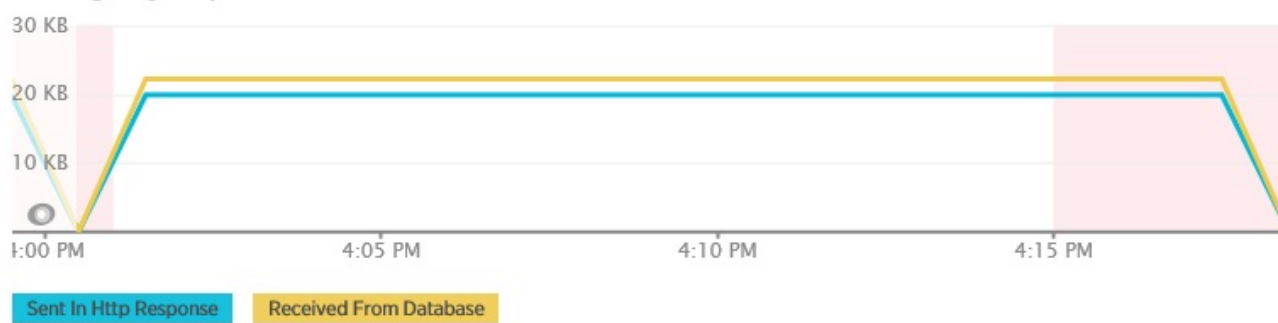


This load test used the same deployment and the same simulated workload of 400 concurrent users as before. The graph shows much lower latency. Response time rises with load to approximately 1.3 seconds, compared to 4 seconds in the previous case. The throughput is also higher at 350 requests per second compared to 100 earlier. The volume of data retrieved from the database now closely matches the size of the HTTP response messages.

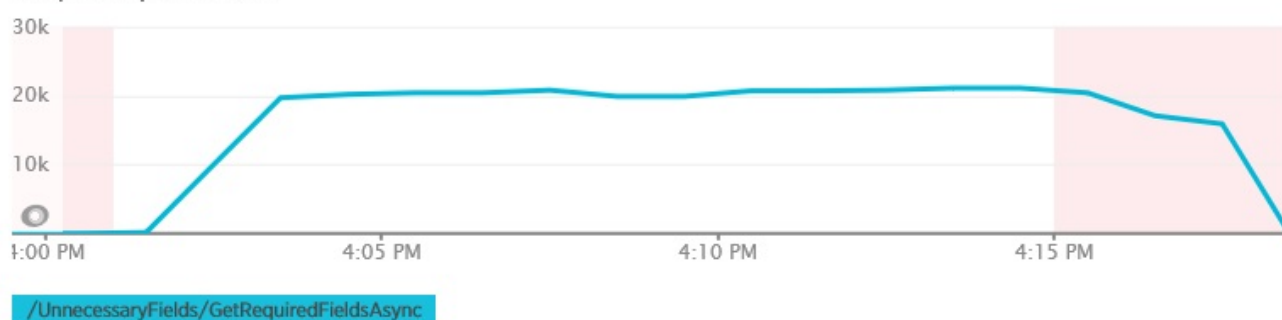
Total Bytes per Minute



Average Bytes per Transaction

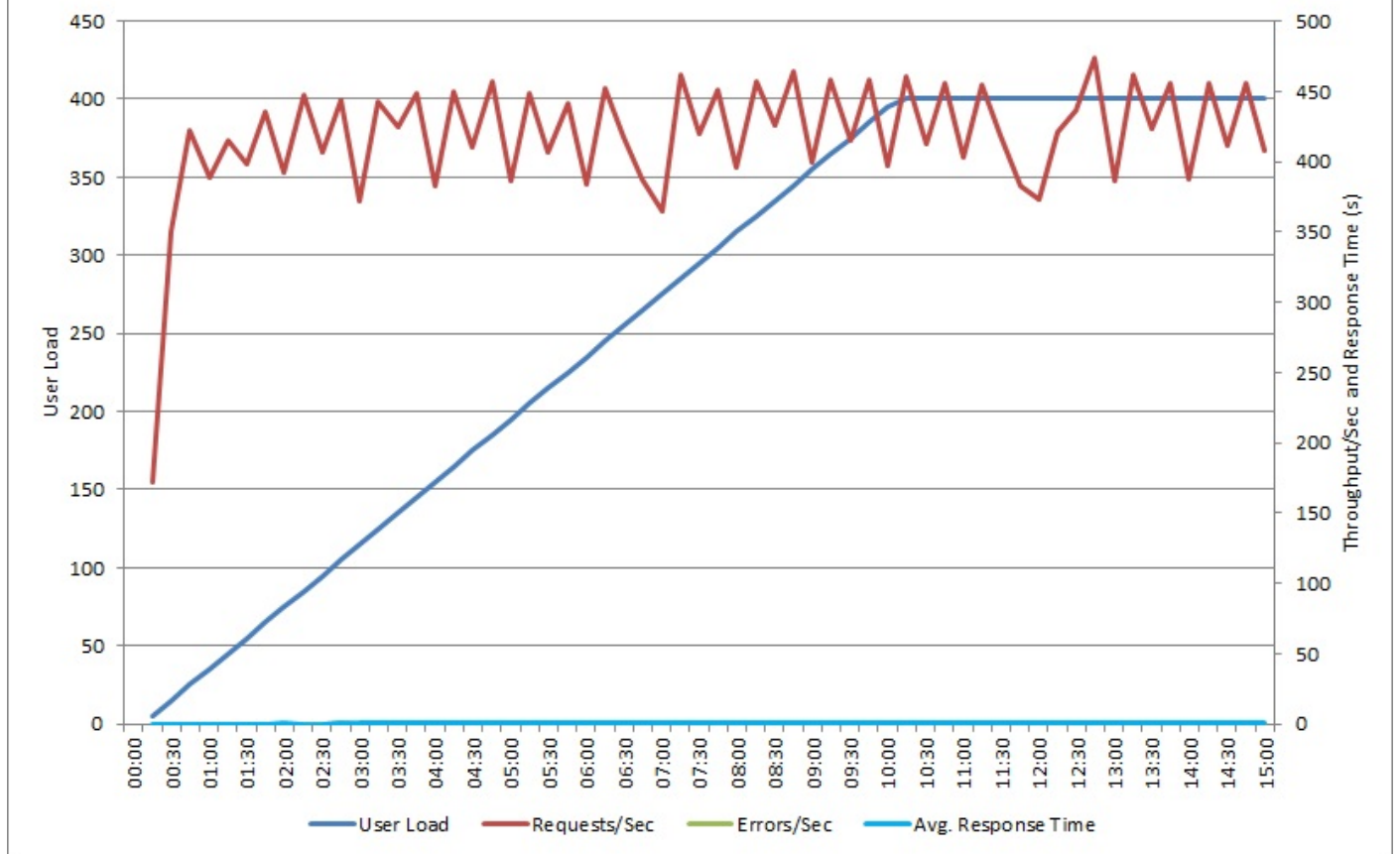


Requests per Minute



Load testing using the `AggregateOnDatabaseAsync` method generates the following results:

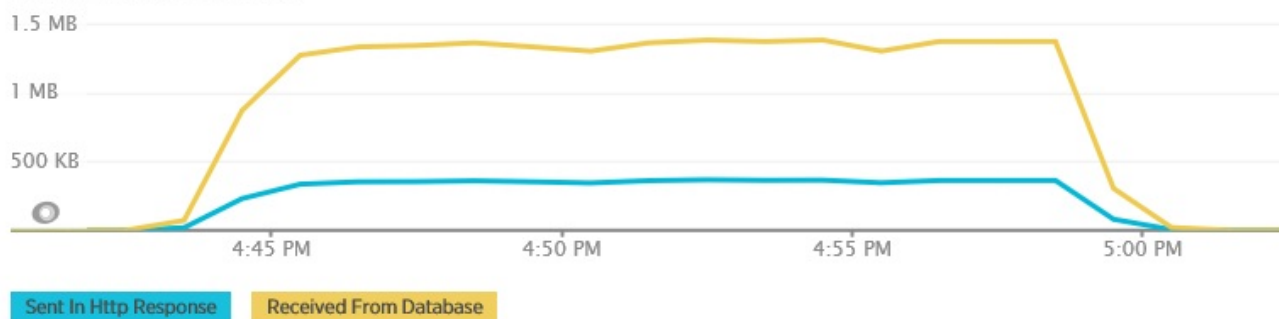
Aggregating In Database



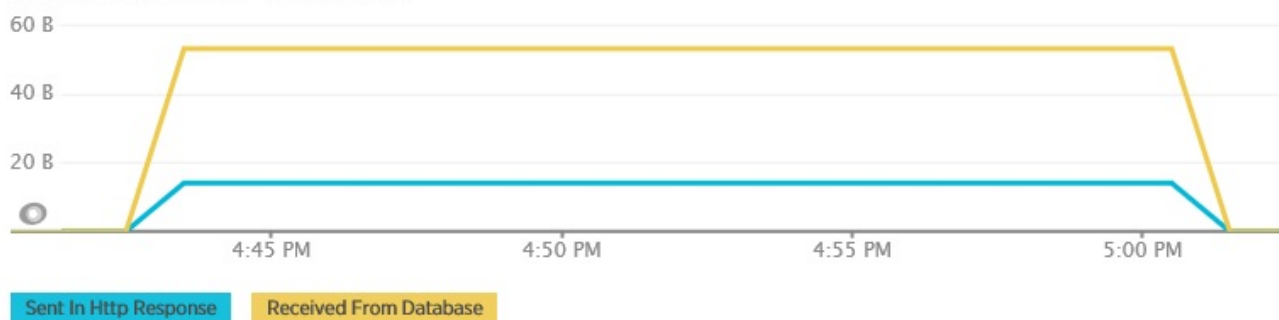
The average response time is now minimal. This is an order of magnitude improvement in performance, caused primarily by the large reduction in I/O from the database.

Here is the corresponding telemetry for the `AggregateOnDatabaseAsync` method. The amount of data retrieved from the database was vastly reduced, from over 280 Kb per transaction to 53 bytes. As a result, the maximum sustained number of requests per minute was raised from around 2,000 to over 25,000.

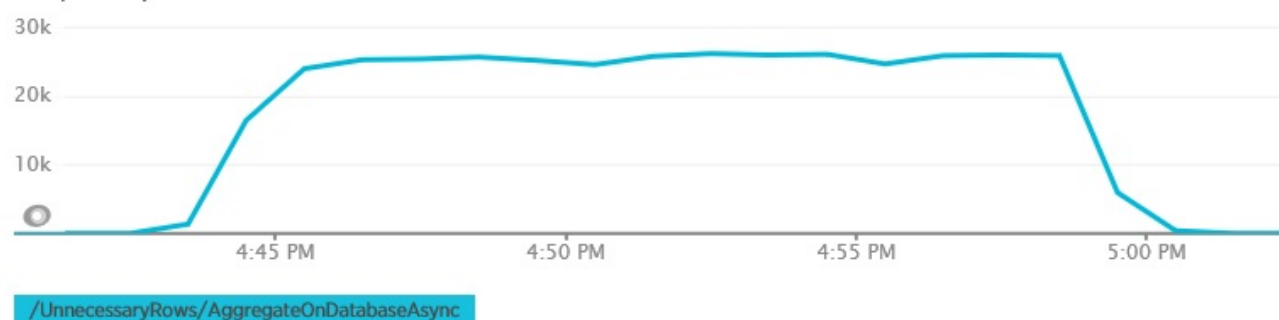
Total Bytes per Minute



Average Bytes per Transaction



Requests per Minute



Related resources

- [Busy Database antipattern](#)
- [Chatty I/O antipattern](#)
- [Data partitioning best practices](#)