Efficient Querying

Article • 01/12/2023

Querying efficiently is a vast subject, that covers subjects as wide-ranging as indexes, related entity loading strategies, and many others. This section details some common themes for making your queries faster, and pitfalls users typically encounter.

Use indexes properly

The main deciding factor in whether a query runs fast or not is whether it will properly utilize indexes where appropriate: databases are typically used to hold large amounts of data, and queries which traverse entire tables are typically sources of serious performance issues. Indexing issues aren't easy to spot, because it isn't immediately obvious whether a given query will use an index or not. For example:

C#

```
// Matches on start, so uses an index (on SQL Server)
var posts1 = context.Posts.Where(p => p.Title.StartsWith("A")).ToList();
// Matches on end, so does not use the index
var posts2 = context.Posts.Where(p => p.Title.EndsWith("A")).ToList();
```

A good way to spot indexing issues is to first pinpoint a slow query, and then examine its query plan via your database's favorite tool; see the performance diagnosis page for more information on how to do that. The query plan displays whether the query traverses the entire table, or uses an index.

As a general rule, there isn't any special EF knowledge to using indexes or diagnosing performance issues related to them; general database knowledge related to indexes is just as relevant to EF applications as to applications not using EF. The following lists some general guidelines to keep in mind when using indexes:

- While indexes speed up queries, they also slow down updates since they need to be kept up-to-date. Avoid defining indexes which aren't needed, and consider using index filters to limit the index to a subset of the rows, thereby reducing this overhead.
- Composite indexes can speed up queries which filter on multiple columns, but they can also speed up queries which don't filter on all the index's columns - depending on ordering. For example, an index on columns A and B speeds up queries filtering by A and B as well as queries filtering only by A, but it does not speed up queries only filtering over B.
- If a query filters by an expression over a column (e.g. `price / 2`), a simple index cannot be used. However, you can define a stored persisted column for your expression, and create an index over that. Some databases also support expression indexes, which can be directly used to speed up queries filtering by any expression.
- Different databases allow indexes to be configured in various ways, and in many cases EF Core providers expose these via the Fluent API. For example, the SQL Server provider allows you to configure whether an index is clustered, or set its fill factor. Consult your provider's documentation for more information.

Project only properties you need

EF Core makes it very easy to query out entity instances, and then use those instances in code. However, querying entity instances can frequently pull back more data than necessary from your database. Consider the following:

C#

```
foreach (var blog in context.Blogs)
{
```

```
    Console.WriteLine("Blog: " + blog.Url);
}
```

Although this code only actually needs each Blog's `Url` property, the entire Blog entity is fetched, and unneeded columns are transferred from the database:

SQL

```
SELECT [b].[BlogId], [b].[CreationDate], [b].[Name], [b].[Rating], [b].[Url]
FROM [Blogs] AS [b]
```

This can be optimized by using `Select` to tell EF which columns to project out:

C#

```
foreach (var blogName in context.Blogs.Select(b => b.Url))
{
    Console.WriteLine("Blog: " + blogName);
}
```

The resulting SQL pulls back only the needed columns:

SQL

```
SELECT [b].[Url]
FROM [Blogs] AS [b]
```

If you need to project out more than one column, project out to a C# anonymous type with the properties you want.

Note that this technique is very useful for read-only queries, but things get more complicated if you need to *update* the fetched blogs, since EF's change tracking only works with entity instances. It's possible to perform updates without loading entire entities by attaching a modified Blog instance and telling EF which properties have changed, but that is a more advanced technique that may not be worth it.

Limit the resultset size

By default, a query returns all rows that matches its filters:

C#

```
var blogsAll = context.Posts
    .Where(p => p.Title.StartsWith("A"))
    .ToList();
```

Since the number of rows returned depends on actual data in your database, it's impossible to know how much data will be loaded from the database, how much memory will be taken up by the results, and how much additional load will be generated when processing these results (e.g. by sending them to a user browser over the network). Crucially, test databases frequently contain little data, so that everything works well while testing, but performance problems suddenly appear when the query starts running on real-world data and many rows are returned.

As a result, it's usually worth giving thought to limiting the number of results:

C#

```
var blogs25 = context.Posts
    .Where(p => p.Title.StartsWith("A"))
    .Take(25)
    .ToList();
```

At a minimum, your UI could show a message indicating that more rows may exist in the database (and allow retrieving them in some other manner). A full-blown solution would implement *pagination*, where your UI only shows a certain number of rows at a time, and allow users to advance to the next page as needed; see the next section for more details on how to implement this efficiently.

Efficient pagination

Pagination refers to retrieving results in pages, rather than all at once; this is typically done for large resultsets, where a user interface is shown that allows the user to navigate to the next or previous page of the results. A common way to implement pagination with databases is to use the `Skip` and `Take` operators (`OFFSET` and `LIMIT` in SQL); while this is an intuitive implementation, it's also quite inefficient. For pagination that allows moving one page at a time (as opposed to jumping to arbitrary pages), consider using *keyset pagination* instead.

For more information, see the documentation page on pagination.

Avoid cartesian explosion when loading related entities

In relational databases, all related entities are loaded by introducing JOINs in single query.

SQL

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId], [p].[AuthorId], [p].[BlogId], [p].
[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
LEFT JOIN [Post] AS [p] ON [b].[BlogId] = [p].[BlogId]
ORDER BY [b].[BlogId], [p].[PostId]
```

If a typical blog has multiple related posts, rows for these posts will duplicate the blog's information. This duplication leads to the so-called "cartesian explosion" problem. As more one-to-many relationships are loaded, the amount of duplicated data may grow and adversely affect the performance of your application.

EF allows avoiding this effect via the use of "split queries", which load the related entities via separate queries. For more information, read the documentation on split and single queries.

> **Note**
>
> The current implementation of **split queries** executes a roundtrip for each query. We plan to improve this in the future, and execute all queries in a single roundtrip.

Load related entities eagerly when possible

It's recommended to read the dedicated page on related entities before continuing with this section.

When dealing with related entities, we usually know in advance what we need to load: a typical example would be loading a certain set of Blogs, along with all their Posts. In these scenarios, it is always better to use eager loading, so that EF can fetch

all the required data in one roundtrip. The filtered include feature also allows you to limit which related entities you'd like to load, while keeping the loading process eager and therefore doable in a single roundtrip:

C#

```
using (var context = new BloggingContext())
{
    var filteredBlogs = context.Blogs
        .Include(
            blog => blog.Posts
                .Where(post => post.BlogId == 1)
                .OrderByDescending(post => post.Title)
                .Take(5))
        .ToList();
}
```

In other scenarios, we may not know which related entity we're going to need before we get its principal entity. For example, when loading some Blog, we may need to consult some other data source - possibly a webservice - in order to know whether we're interested in that Blog's Posts. In these cases, explicit or lazy loading can be used to fetch related entities separately, and populate the Blog's Posts navigation. Note that since these methods aren't eager, they require additional roundtrips to the database, which is source of slowdown; depending on your specific scenario, it may be more efficient to just always load all Posts, rather than to execute the additional roundtrips and selectively get only the Posts you need.

Beware of lazy loading

Lazy loading often seems like a very useful way to write database logic, since EF Core automatically loads related entities from the database as they are accessed by your code. This avoids loading related entities that aren't needed (like explicit loading), and seemingly frees the programmer from having to deal with related entities altogether. However, lazy loading is particularly prone for producing unneeded extra roundtrips which can slow the application.

Consider the following:

C#

```
foreach (var blog in context.Blogs.ToList())
{
    foreach (var post in blog.Posts)
    {
        Console.WriteLine($"Blog {blog.Url}, Post: {post.Title}");
    }
}
```

This seemingly innocent piece of code iterates through all the blogs and their posts, printing them out. Turning on EF Core's statement logging reveals the following:

Console

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [b].[BlogId], [b].[Rating], [b].[Url]
      FROM [Blogs] AS [b]
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (5ms) [Parameters=[@__p_0='1'], CommandType='Text', CommandTimeout='30']
      SELECT [p].[PostId], [p].[BlogId], [p].[Content], [p].[Title]
      FROM [Post] AS [p]
      WHERE [p].[BlogId] = @__p_0
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[@__p_0='2'], CommandType='Text', CommandTimeout='30']
```

```
      SELECT [p].[PostId], [p].[BlogId], [p].[Content], [p].[Title]
      FROM [Post] AS [p]
      WHERE [p].[BlogId] = @__p_0
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[@__p_0='3'], CommandType='Text', CommandTimeout='30']
      SELECT [p].[PostId], [p].[BlogId], [p].[Content], [p].[Title]
      FROM [Post] AS [p]
      WHERE [p].[BlogId] = @__p_0

... and so on
```

What's going on here? Why are all these queries being sent for the simple loops above? With lazy loading, a Blog's Posts are only (lazily) loaded when its Posts property is accessed; as a result, each iteration in the inner foreach triggers an additional database query, in its own roundtrip. As a result, after the initial query loading all the blogs, we then have another query *per blog*, loading all its posts; this is sometimes called the *N+1* problem, and it can cause very significant performance issues.

Assuming we're going to need all of the blogs' posts, it makes sense to use eager loading here instead. We can use the Include operator to perform the loading, but since we only need the Blogs' URLs (and we should only load what's needed). So we'll use a projection instead:

C#

```
foreach (var blog in context.Blogs.Select(b => new { b.Url, b.Posts }).ToList())
{
    foreach (var post in blog.Posts)
    {
        Console.WriteLine($"Blog {blog.Url}, Post: {post.Title}");
    }
}
```

This will make EF Core fetch all the Blogs - along with their Posts - in a single query. In some cases, it may also be useful to avoid cartesian explosion effects by using split queries.

> ⚠ **Warning**
>
> Because lazy loading makes it extremely easy to inadvertently trigger the N+1 problem, it is recommended to avoid it. Eager or explicit loading make it very clear in the source code when a database roundtrip occurs.

Buffering and streaming

Buffering refers to loading all your query results into memory, whereas streaming means that EF hands the application a single result each time, never containing the entire resultset in memory. In principle, the memory requirements of a streaming query are fixed - they are the same whether the query returns 1 row or 1000; a buffering query, on the other hand, requires more memory the more rows are returned. For queries that result large resultsets, this can be an important performance factor.

Whether a query buffers or streams depends on how it is evaluated:

C#

```
// ToList and ToArray cause the entire resultset to be buffered:
var blogsList = context.Posts.Where(p => p.Title.StartsWith("A")).ToList();
var blogsArray = context.Posts.Where(p => p.Title.StartsWith("A")).ToArray();

// Foreach streams, processing one row at a time:
foreach (var blog in context.Posts.Where(p => p.Title.StartsWith("A")))
```

```
{
    // ...
}

// AsEnumerable also streams, allowing you to execute LINQ operators on the client-side:
var doubleFilteredBlogs = context.Posts
    .Where(p => p.Title.StartsWith("A")) // Translated to SQL and executed in the database
    .AsEnumerable()
    .Where(p => SomeDotNetMethod(p)); // Executed at the client on all database results
```

If your queries return just a few results, then you probably don't have to worry about this. However, if your query might return large numbers of rows, it's worth giving thought to streaming instead of buffering.

> **Note**
>
> Avoid using **ToList** or **ToArray** if you intend to use another LINQ operator on the result - this will needlessly buffer all results into memory. Use **AsEnumerable** instead.

Internal buffering by EF

In certain situations, EF will itself buffer the resultset internally, regardless of how you evaluate your query. The two cases where this happens are:

- When a retrying execution strategy is in place. This is done to make sure the same results are returned if the query is retried later.
- When split query is used, the resultsets of all but the last query are buffered - unless MARS (Multiple Active Result Sets) is enabled on SQL Server. This is because it is usually impossible to have multiple query resultsets active at the same time.

Note that this internal buffering occurs in addition to any buffering you cause via LINQ operators. For example, if you use ToList on a query and a retrying execution strategy is in place, the resultset is loaded into memory *twice*: once internally by EF, and once by ToList.

Tracking, no-tracking and identity resolution

It's recommended to read the dedicated page on tracking and no-tracking before continuing with this section.

EF tracks entity instances by default, so that changes on them are detected and persisted when SaveChanges is called. Another effect of tracking queries is that EF detects if an instance has already been loaded for your data, and will automatically return that tracked instance rather than returning a new one; this is called *identity resolution*. From a performance perspective, change tracking means the following:

- EF internally maintains a dictionary of tracked instances. When new data is loaded, EF checks the dictionary to see if an instance is already tracked for that entity's key (identity resolution). The dictionary maintenance and lookups take up some time when loading the query's results.
- Before handing a loaded instance to the application, EF *snapshots* that instance and keeps the snapshot internally. When SaveChanges is called, the application's instance is compared with the snapshot to discover the changes to be persisted. The snapshot takes up more memory, and the snapshotting process itself takes time; it's sometimes possible to specify different, possibly more efficient snapshotting behavior via value comparers, or to use change-tracking proxies to bypass the snapshotting process altogether (though that comes with its own set of disadvantages).

In read-only scenarios where changes aren't saved back to the database, the above overheads can be avoided by using no-tracking queries. However, since no-tracking queries do not perform identity resolution, a database row which is referenced

by multiple other loaded rows will be materialized as different instances.

To illustrate, assume we are loading a large number of Posts from the database, as well as the Blog referenced by each Post. If 100 Posts happen to reference the same Blog, a tracking query detects this via identity resolution, and all Post instances will refer the same de-duplicated Blog instance. A no-tracking query, in contrast, duplicates the same Blog 100 times - and application code must be written accordingly.

Here are the results for a benchmark comparing tracking vs. no-tracking behavior for a query loading 10 Blogs with 20 Posts each. The source code is available here , feel free to use it as a basis for your own measurements.

Expand table

Method	NumBlogs	NumPostsPerBlog	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
AsTracking	10	20	1,414.7 us	27.20 us	45.44 us	1,405.5 us	1.00	0.00	60.5469	13.6719	-	380.11 KB
AsNoTracking	10	20	993.3 us	24.04 us	65.40 us	966.2 us	0.71	0.05	37.1094	6.8359	-	232.89 KB

Finally, it is possible to perform updates without the overhead of change tracking, by utilizing a no-tracking query and then attaching the returned instance to the context, specifying which changes are to be made. This transfers the burden of change tracking from EF to the user, and should only be attempted if the change tracking overhead has been shown to be unacceptable via profiling or benchmarking.

Using SQL queries

In some cases, more optimized SQL exists for your query, which EF does not generate. This can happen when the SQL construct is an extension specific to your database that's unsupported, or simply because EF does not translate to it yet. In these cases, writing SQL by hand can provide a substantial performance boost, and EF supports several ways to do this.

- Use SQL queries directly in your query, e.g. via FromSqlRaw. EF even lets you compose over the SQL with regular LINQ queries, allowing you to express only a part of the query in SQL. This is a good technique when the SQL only needs to be used in a single query in your codebase.
- Define a user-defined function (UDF), and then call that from your queries. Note that EF allows UDFs to return full resultsets - these are known as table-valued functions (TVFs) - and also allows mapping a `DbSet` to a function, making it look just like just another table.
- Define a database view and query from it in your queries. Note that unlike functions, views cannot accept parameters.

> **Note**
>
> Raw SQL should generally be used as a last resort, after making sure that EF can't generate the SQL you want, and when performance is important enough for the given query to justify it. Using raw SQL brings considerable maintenance disadvantages.

Asynchronous programming

As a general rule, in order for your application to be scalable, it's important to always use asynchronous APIs rather than synchronous one (e.g. SaveChangesAsync rather than SaveChanges). Synchronous APIs block the thread for the duration of database I/O, increasing the need for threads and the number of thread context switches that must occur.

For more information, see the page on async programming.

> ⚠ **Warning**
>
> Avoid mixing synchronous and asynchronous code in the same application - it's very easy to inadvertently trigger subtle thread-pool starvation issues.

> ⚠ **Warning**
>
> The async implementation of Microsoft.Data.SqlClient unfortunately has some known issues (e.g. #593 , #601 , and others). If you're seeing unexpected performance problems, try using sync command execution instead, especially when dealing with large text or binary values.

Additional resources

- See the advanced performance topics page for additional topics related to efficient querying.
- See the performance section of the null comparison documentation page for some best practices when comparing nullable values.