Imagine you're generating a report like this:

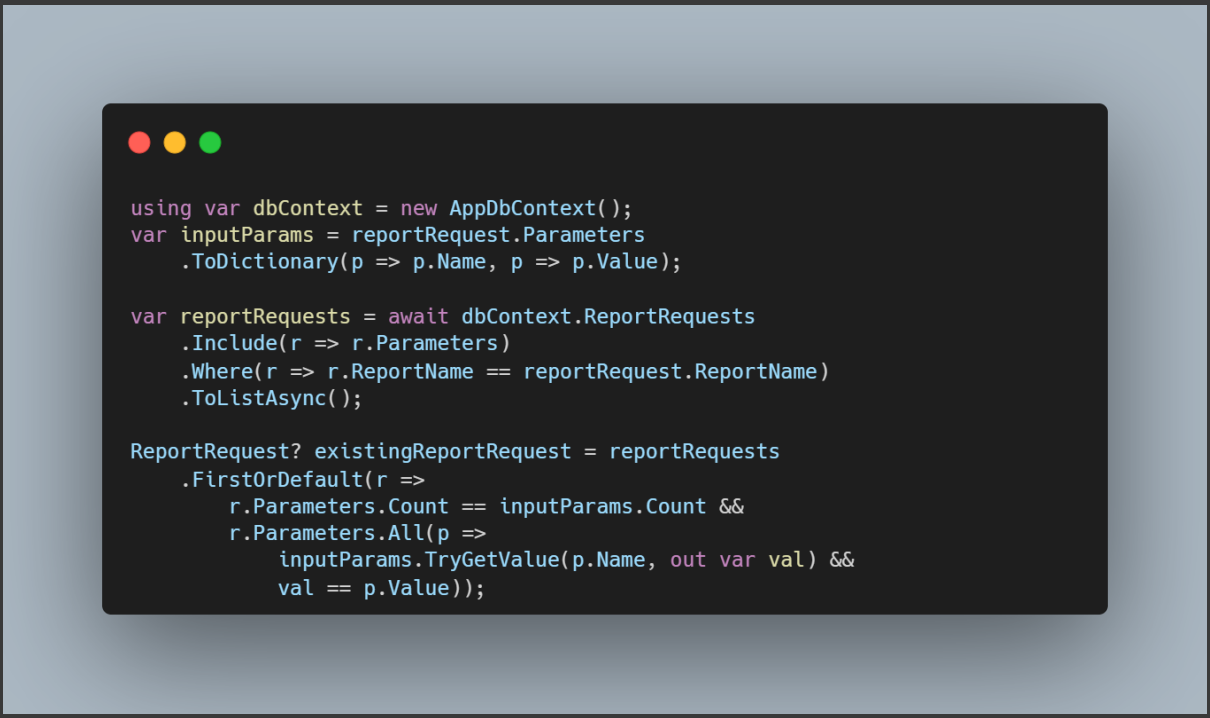
```
ReportRequest reportRequest = ReportRequest.Create("SalesReport",
    new Dictionary<string, string>
    {
        { "StartDate", "2023-01-01" },
        { "EndDate", "2023-12-31" },
        { "Region", "North America" },
        { "Category", "Electronics" },
        { "Format", "PDF" },
        { "IncludeCharts", "true" },
        { "IncludeSummary", "true" },
        { "Language", "en-US" },
        { "Timezone", "America/New_York" },
        { "Currency", "USD" },
        { "CustomerType", "Retail" },
        { "SalesChannel", "Online" },
        { "IncludeDiscounts", "true" },
        { "IncludeReturns", "false" },
        { "SortBy", "Date" },
        { "SortOrder", "Descending" }
    });
```

```
public class ReportRequest
{
    private ReportRequest() { }
    private ReportRequest(string reportName, Dictionary<string, string> parameters)
    {
        ReportName = reportName;
        Parameters = [.. parameters.
            Select(kvp => new ReportRequestParameter(kvp.Key, kvp.Value))];
    }

    public Guid Id { get; private set; } = Guid.NewGuid();
    public string ReportName { get; private set; } = string.Empty;
    public List<ReportRequestParameter> Parameters { get; private set; } = [];

    public static ReportRequest Create(string reportName, Dictionary<string, string> parameters)
        => new(reportName, parameters);
}

public class ReportRequestParameter
{
    private ReportRequestParameter() { }
    public ReportRequestParameter(string name, string value)
    {
        Name = name;
        Value = value;
    }

    public Guid Id { get; private set; } = Guid.NewGuid();
    public Guid ReportRequestId { get; private set; }
    public string Name { get; private set; } = string.Empty;
    public string Value { get; private set; } = string.Empty;
}
```

Seems straightforward — until you want to **check if the same report was already generated**.

😬 The Matching Nightmare

You now need to query your DB like this: to find duplicate or to reuse already generated report.



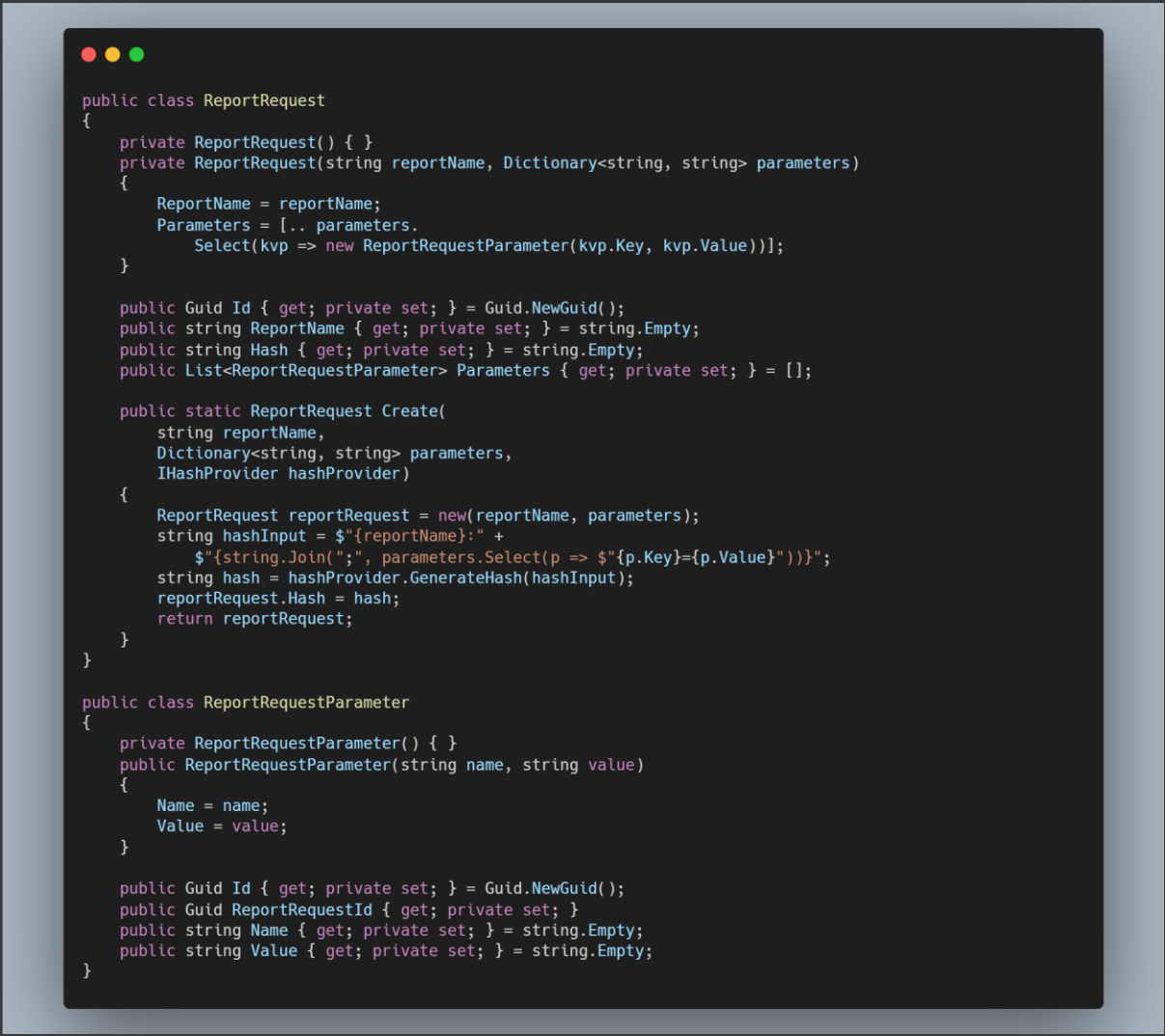
```
using var dbContext = new AppDbContext();
var inputParams = reportRequest.Parameters
    .ToDictionary(p => p.Name, p => p.Value);

var reportRequests = await dbContext.ReportRequests
    .Include(r => r.Parameters)
    .Where(r => r.ReportName == reportRequest.ReportName)
    .ToListAsync();

ReportRequest? existingReportRequest = reportRequests
    .FirstOrDefault(r =>
        r.Parameters.Count == inputParams.Count &&
        r.Parameters.All(p =>
            inputParams.TryGetValue(p.Name, out var val) &&
            val == p.Value));
```

That’s:

- In-memory filtering
- Complex LINQ logic
- No real indexing
- Slower as data grows

✅ The Better Way: Use a Hash

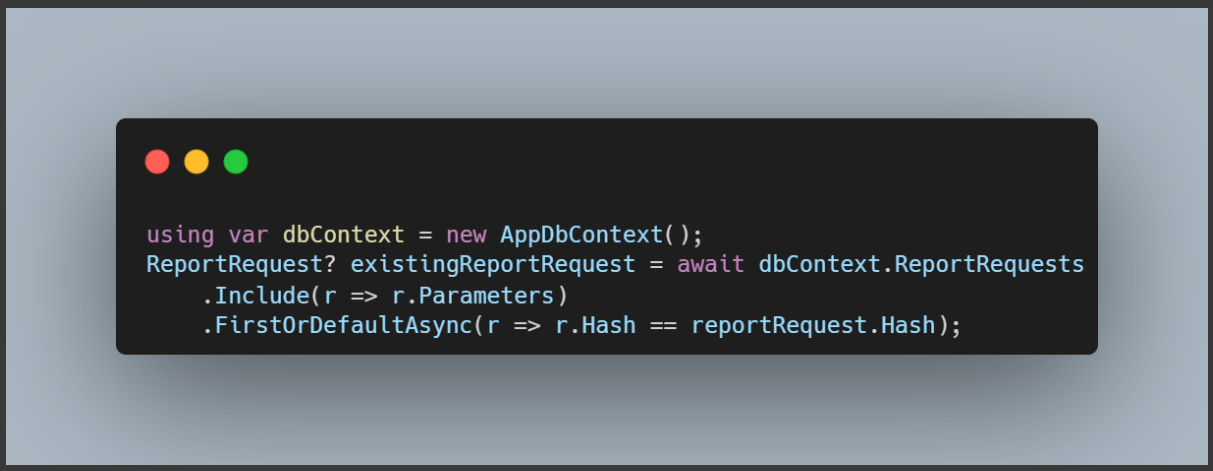
Instead of comparing all 15+ parameters each time, generate a **deterministic hash** like this:



```
public class ReportRequest
{
    private ReportRequest() { }
    private ReportRequest(string reportName, Dictionary<string, string> parameters)
    {
        ReportName = reportName;
        Parameters = [.. parameters.
            Select(kvp => new ReportRequestParameter(kvp.Key, kvp.Value))];
    }

    public Guid Id { get; private set; } = Guid.NewGuid();
    public string ReportName { get; private set; } = string.Empty;
    public string Hash { get; private set; } = string.Empty;
    public List<ReportRequestParameter> Parameters { get; private set; } = [];

    public static ReportRequest Create(
        string reportName,
        Dictionary<string, string> parameters,
        IHashProvider hashProvider)
    {
        ReportRequest reportRequest = new(reportName, parameters);
        string hashInput = $"{reportName}:" +
            $"{string.Join(";", parameters.Select(p => $"{p.Key}={p.Value}"))}";
        string hash = hashProvider.GenerateHash(hashInput);
        reportRequest.Hash = hash;
        return reportRequest;
    }
}

public class ReportRequestParameter
{
    private ReportRequestParameter() { }
    public ReportRequestParameter(string name, string value)
    {
        Name = name;
        Value = value;
    }

    public Guid Id { get; private set; } = Guid.NewGuid();
    public Guid ReportRequestId { get; private set; }
    public string Name { get; private set; } = string.Empty;
    public string Value { get; private set; } = string.Empty;
}
```

Store the hash with each request.

Now your query becomes:



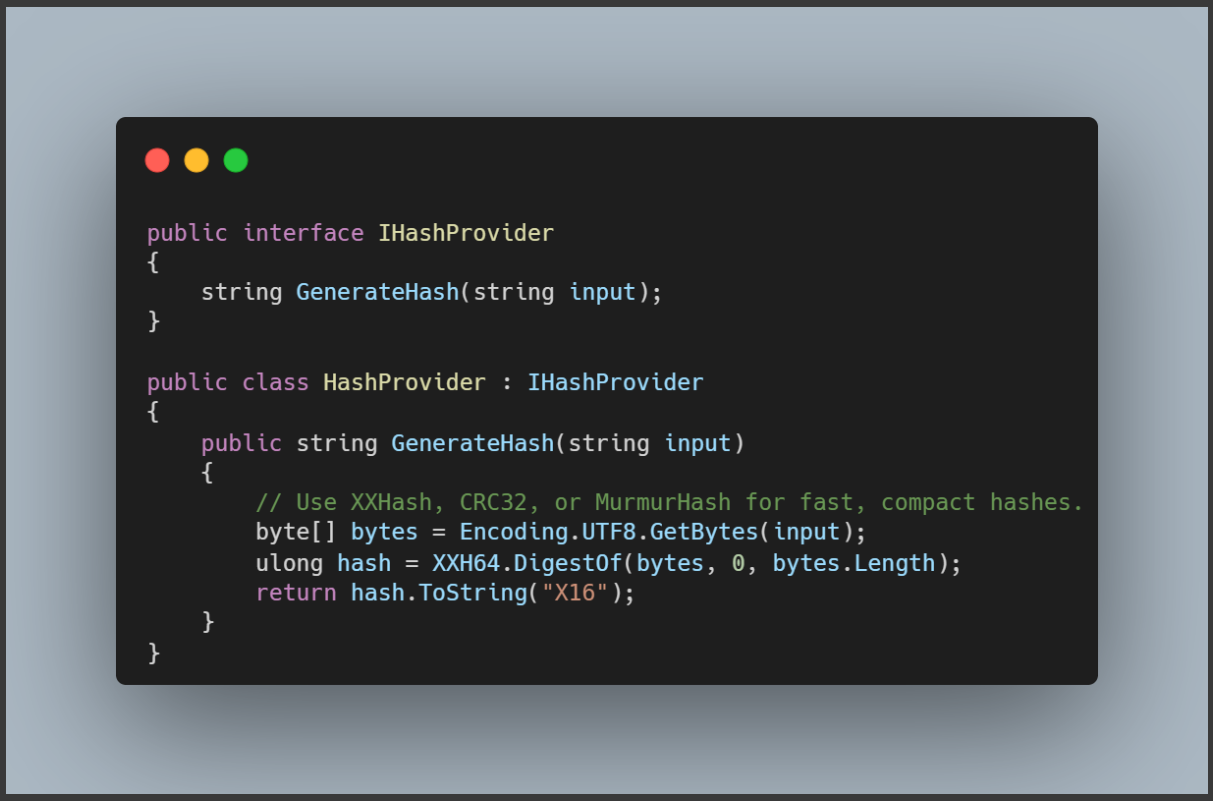
```
using var dbContext = new AppDbContext();
ReportRequest? existingReportRequest = await dbContext.ReportRequests
    .Include(r => r.Parameters)
    .FirstOrDefaultAsync(r => r.Hash == reportRequest.Hash);
```

⚡ Fast
⚡ Indexable
⚡ Simple

💡 **If you're matching on many fields or rows — whether it's reports, payments, scheduling, or config — hashing the input can dramatically simplify your logic.**

🔁 Stop comparing all the fields. Start comparing the hash.

Hashing provider which I used XXHash,



```
public interface IHashProvider
{
    string GenerateHash(string input);
}

public class HashProvider : IHashProvider
{
    public string GenerateHash(string input)
    {
        // Use XXHash, CRC32, or MurmurHash for fast, compact hashes.
        byte[] bytes = Encoding.UTF8.GetBytes(input);
        ulong hash = XXH64.DigestOf(bytes, 0, bytes.Length);
        return hash.ToString("X16");
    }
}
```

Use Case	Best Choice	Output Length	Notes
Cryptographic security	SHA256 (full)	64 hex / 44 b64	Strong, but large
Fast + compact fingerprint	CRC32 or XXHash	8–10 chars	Fast, good for duplicates
Safe + moderately short	Truncated SHA256	8–16 chars	Low collision chance, compact
Compact + alphanumeric	Base62(SHA256)	~22 chars	Very short and safe if encoded well