Spock's Disciples

Group members:

Lars, Oskar, Maxime, and Frode

IKT453-G 24V

Intelligent Data Management

Faculty of Technology and Natural Sciences

Universitetet of Agder

Grimstad, April 2024

Contents

List of Figures

List of Tables

Listings

1 Introduction

In this project we have been tasked to create a data warehouse consisting of three databases, one primary which consists of a SQL rational database and two secondary that consists of no-SQL databases. The rational database we choose as our primary was the framework PostgreSQL. We choose this framework because of its capabilities with complex queries and its simplicity in implementation. Furthermore our two choices for secondary databases was the no-SQL databases MongoDB and Redis. We choose MongoDB because of ts flexibility and it was recommended to be used by our professor. The last no-SQL database Redis was chosen as a substitute of a previous chosen database, which was Cassandra. It was chosen as a substitute because of its possible speed and simplicity for caching and real-time operations.

It was decided to use an E-commercial dataset to populate and find queries in our databases. following this we decided to use an ASP .NET approach to our website, where we would be hosting a connection to the three described databases. The databases were chosen for use through the Controller-Model-View framework that .NET provides. This was then built and run using docker images.

1.1 Pre-implementation assumptions about the system

When dealing with three different kinds of databases in a data warehouse, a number of assumptions are made before implementation. For example, one assumption we made was that there would be an increasing need for pre-aggregation of queries with the no-SQL databases. We based this assumption on the schema-less nature of our no-SQL databases, anticipating that pre-aggregated data could enhance query performance by adapting to the flexible structure of no-SQL databases.

Another assumption we had was that the data models of PostgreSQL (relational), MongoDB (document), and Redis (key-value) can be quite easy to integrated or transformed to provide a unified view of data (As we have discovered later to be more tricky).

Another assumption that we made was that PostgreSQL would be dealing with the data better considering the dataset was more structured and relational database friendly. We also assumed PostgreSQL would be more flexible and capable of complexity in its queries based on relational database attributes and SQL querying.

Some other assumptions we had based on our knowledge of SQL and no-SQL databases was that the no-SQL databases (MongoDB and Redis) could offer better performance and scalability. Some of its performance improvement we assumed would come from its pre-aggravation of queries, rather than the joint and grouping operation of the PostgreSQL relational database system. (This was a mixed assumption)

Furthermore in to the assumption of the implementation, we assumed integration of these databases within an ASP.NET MVC application would be straightforward, assuming the Entity Framework for PostgreSQL and custom or third-party libraries for MongoDB and Redis could abstract most of the complexity. We also assumed the MVC architecture would cleanly separate the concerns, assuming database interactions would be encapsulated within the model layer, and queries would be done through the controller view connection.

2 Theoretical background

2.1 Integration of SQL and NoSQL Databases

Integrating SQL and NoSQL databases capitalizes on the capabilities of both types of systems, which is an important topic in data management systems[4]. SQL databases such as PostgreSQL, excel at performing sophisticated queries and maintaining structured data. On the other hand, NoSQL databases like MongoDB and Redis, are prized for their flexibility and speed, making them ideal for unstructured data and real-time processes. the paper[5] describes combining these technologies to meet a wide range of data storage and processing requirements, which could be done using each system's unique capabilities.

Hybrid Database Models Hybrid models have the benefits of both SQL and NoSQL databases, combining their features with specific data handling needs:

- **SQL Databases** — Typically used for managing transactional data, where keeping things accurate and consistent is super important. This means that whenever dealing with operations like banking transactions, booking systems, or anything where it's essential that the data remains perfectly in sync and error-free, SQL databases are the go-to. it's designed to handle complex queries and ensure that every data interaction follows strict rules, so nothing falls through the cracks.
- **NoSQL Databases** — These are the workhorses to choose when dealing with massive amounts of data that don't fit neatly into traditional table schemas, like social media posts, sensor data, or real-time analytics. NoSQL databases are all about flexibility and speed, allowing you to quickly process and analyze different types of data. They're perfect for situations where you need to scale rapidly and handle new data types on the fly without disrupting existing operations.

2.2 PostgreSQL

PostgreSQL is a powerful, enterprise-class open-source relational database that allows for both SQL (relational) and JSON (non-relational) queries[10]. With over 20 years of community development under its belt, it is an extremely reliable database management system. Its high degrees of correctness, durability, and integrity are a result of this careful and cooperative process. Many web, mobile, geospatial, and analytics applications use PostgreSQL as their primary data storage or data warehouse.

Key Features

- **ACID Compliance:** — ACID refers to atomicity, consistency, isolation, and durability—four characteristics that make database transactions reliable. More specifically, ACID says that since partial modifications were never saved, the data in a database is accurate. PostgreSQL features that enable ACID compliance include write-ahead logging, Multi-Version Concurrency Control (MVCC), and point-in-time recovery.
- **Advanced SQL Support:** — With support for complex queries, foreign keys, triggers, and more, PostgreSQL is capable of handling sophisticated data operations, making it ideal for applications that require deep data manipulation.

- **Extensibility and PL/pgSQL:** — Customizability is a key strength of PostgreSQL. It supports a wide range of programming languages for procedural coding including Python, Java, Perl, .Net, Go, Ruby, C/C++, Tcl, and ODBC, enabling highly customized and optimized applications.

PostgreSQL excels in online transaction processing (OLTP) because it supports automated failover and full redundancy. It is frequently used in complex web applications that need strong back-end support as well as financial systems, which have strict requirements for correctness and consistency. [10]

The Difference with a relational database is where traditionally, the RD is a collection of data elements having predetermined relationships. These objects are grouped into tables with columns and rows. Software that enables reading, writing, and modifying relational databases is known as a relational database management system PostgreSQL is an object-relational database management system (ORDMBS), which implies it combines relational capabilities with an object-oriented design.

2.3 MongoDB

The scale-out architecture upon which MongoDB is based has gained popularity among developers for creating scalable systems with dynamic data structures. Developers may easily store structured and unstructured data with MongoDB because it is a document database. It stores documents in a format resembling JSON.[11] Since developers do not have to worry about normalizing data, this format is a suitable choice because it maps to native objects in the majority of contemporary programming languages. In order to meet massive data loads, MongoDB can scale both vertically and horizontally. It can also handle enormous volumes.

This model is particularly advantageous for applications that require agility and the ability to handle large volumes of data without the constraints of traditional relational schemas. MongoDB was designed for web developers and business application architects who need to build applications that expand gracefully and rapidly. MongoDB is used by development teams and companies of all sizes for a multitude of purposes.[11]

Key Features

- **Schema-less:** Data can be stored in flexible JSON-like formats with MongoDB, enabling unique structures for each document.
- **Sharding:** Sharding is a method for distributing data across multiple machines.[8] This technique is used by MongoDB to facilitate high throughput operations and deployments of very large data sets. Large data volumes or high throughput applications in database systems can exceed a single server's capability. For instance, a server's CPU capacity may be depleted by excessive query rates. Operating set sizes that exceed the RAM of the system put a strain on disk drives' I/O capabilities.[8]
- **Replication:** A collection of 'mongod' processes that preserve the same data set is known as a replica set in MongoDB[7]. Replica sets are the foundation of all production deployments, offering high availability and redundancy. With multiple copies of data on different database

servers, replication provides a level of fault tolerance against the loss of a single database server.

Applications MongoDB excels in environments that demand high-speed operations and scalable storage:

- **Mobile Applications:** For mobile platforms, where user data and interaction logs are varied and voluminous, MongoDB provides the scalability needed to handle sudden spikes in data traffic and storage.
- **Content Management Systems:** Its dynamic schema caters well to content management systems, which often need to manage a diverse set of content types and user-generated data with minimal latency.
- **Real-Time Analytics:** MongoDB is well-suited for real-time analytics applications, where its ability to perform fast reads and writes can significantly enhance the responsiveness of data analysis tools.

2.4 Redis

Redis is an in-memory data store that millions of developers use for caching, vector databases, document databases, streaming engines, and message brokers.[**redis4**, 1] Redis includes built-in replication and varying levels of on-disk persistence. It supports complicated data types (such as strings, hashes, lists, sets, sorted sets, and JSON) and defines atomic actions on them.

Key Features

- **In-Memory Storage:** Redis is a database that runs mostly in main memory and offers extremely rapid data access compared to disk-based databases. Because of this capability, it is a great option for caching since it minimizes the need to often query slower backend databases.[6]
- **Data Structures:** Redis offers advanced data structures such strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, and geospatial indexes, in contrast to typical key-value stores. Redis may as such, be used by developers for tasks such as straightforward caching, or programming tasks like queues and real-time counting systems.[6]

  Redis also works to improve application performance and ensure that data management is done quickly and skillfully. Redis offers decent performance and dependability for most types of application, be it a social networking platform handling millions of user interactions or a financial trading platform requiring real-time data updates.

2.5 Comparison between relational and NoSQL databases

Relational databases use strict schemas, constraints, keys, and relationships between tables to arrange data into tables with rows and columns. Because these systems are structured, they can handle intricate queries and transactions while maintaining data integrity because of their ACID (Atomicity, Consistency, Isolation, Durability) features. However, because of their schemas' rigidity, it might be difficult to adjust to new or changing data requirements, possibly needing large changes and downtime.

Document-oriented NoSQL databases, such as MongoDB, provide a more flexible structure by storing data in documents that resemble JSON. Because of its adaptability, complex and hierarchical data structures can be stored in an intelligible manner and can be readily modified without requiring significant reworking. However a lack of standards can result in redundant and inconsistent data, and maintaining data consistency and integrity frequently calls for more manual labor.

The most basic type of NoSQL database is a key-value store, like Redis, which arranges data as collections of key-value pairs. Because of their incredibly quick read and write speeds, they are perfect for real-time applications, session storage, and caching. They are less fitting for applications needing complicated data connections because of their intrinsic lack of data relationship management and limited functionality for complex queries.

Specific application requirements, such as the type of data, query complexity, scalability requirements, and data integrity significance, should guide the decision between relational and NoSQL databases. Applications that demand precise data integrity and complex querying capabilities are better served by relational databases. While key-value stores are ideal for applications that prioritize speed and scalability with straightforward data structures, document-oriented NoSQL databases are well-suited for scenarios demanding schema flexibility and quick development. The choice of database type to utilize is based on matching application requirements with database capabilities. Different database types fulfill different purposes.

2.6 State of the art

Database integration research tries to address difficulties such as data consistency, synchronization, and query performance across different database systems[4]. Because middleware solutions and data abstraction layers manage the complexities of cross-database queries and transactions, they improve the scalability and efficiency of data systems. this is why they are often explored.

Data Federation Techniques A data federation is a software process that allows multiple databases to function as one.[9] This virtual database collects data from a variety of sources and converts it to a single model. This allows front-end applications to access a single source of data.

It is part of the data virtualization framework. This data virtualization expanded with data federation, but it also introduced new features, applications, and functionalities. As a result, data virtualization can be utilized for a wide range of applications other than data warehouse compilation. It includes metadata repositories, data abstraction, read/write access to source data systems, and advanced security. Data federation is a component of data virtualization, however, the two are not synonymous.

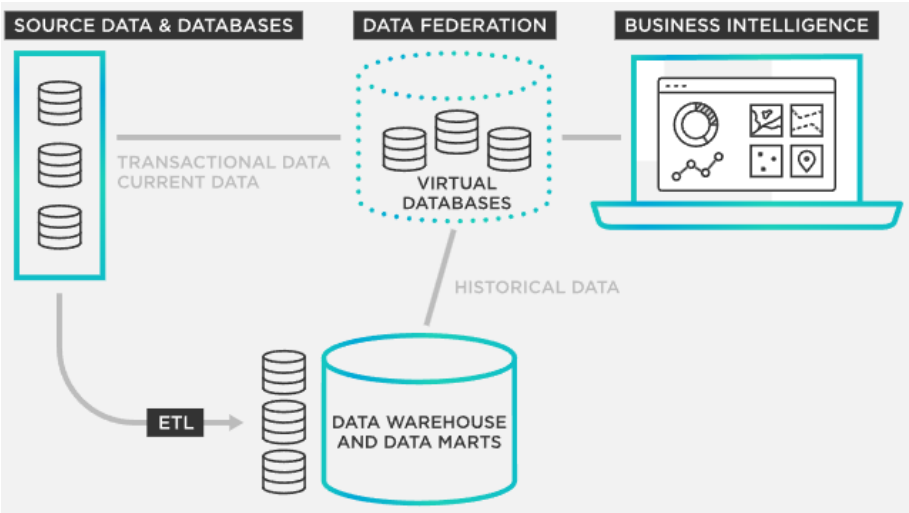


Figure 1: An example of a data federation setup

Unified Data Access Layers Consider the unified data access layer to be a translator, allowing everyone to communicate in the same language and smoothing the communications between databases. It functions as a bridge, transforming queries into something all backend systems can understand, regardless of their original dialects.

- **Utilizes Modern Technologies** — This layer uses innovative tools like OData, GraphQL, or custom APIs to ensure that no matter how data is stored across your systems, the applications may communicate with each database in a consistent manner. Whether it needs to run a fast report or change information across many systems, these technologies assist in standardizing how you request and manage data.
- **Enhances Maintainability** — Because this layer handles all of the complexities of working with many databases, the rest of the application may relax. This implies that if the database team needs to make a change to the storage layer, the app will be unaffected. The single layer ensures that the application operates consistently, making maintenance and upgrading much easier.

2.6.1 Designing Solution Architectures

In theoretical frameworks, using ASP.NET MVC to handle connected databases is considered a strategic solution. Using Entity Framework with PostgreSQL, as well as custom or third-party MongoDB and Redis modules, can help to reduce many of the intricacies of database interfaces. The MVC architecture encourages a clear separation of responsibilities by isolating database operations within the model layer, while the controller orchestrates data flow and the view presents it, resulting in a simplified and modular system architecture.

2.6.2 ASP.NET Core

ASP.NET Core is an open-source cross-platform framework developed by Microsoft. It is designed for modern cloud and web application creation. One of the strengths of ASP.NET Core is its performance and scalability. It can handle large amounts of data, supports asynchronous programming which allows for efficient management of multiple requests simultaneously, and can be hosted on various server environments.

Through the Entity Framework, ASP.NET Core offers a robust database connection which supports many database systems, including:

- SQL Server
- MySQL
- Oracle

The Framework is also designed to build API's, as it is oriented towards making communication and information flow between front-end applications and back-end databases as easy as possible.

Various security features, including support for:

- Data protection
- SSL enforcement
- CORS management
- Authentication
- Authorization

are included in ASP.NET Core. In the context of a data warehouse, these characteristics are vital for guaranteeing safe data transactions. Furthermore, the framework provides a modular architecture that enables application developers to create micro-services architectures. This can be helpful for managing complex data operations and scaling applications, which is commonly needed in data warehouse scenarios.

3 Method and Implementation

3.1 Docker File and Docker hub

The way we chose to do this was by creating an ASP.net core default project then editing the ‘launchsettings.json‘ to allow Docker to port it to our localhost using the 5000/5001 port, see figure 2 and in the ‘docker-compose.yml‘ file 4.

Figure 2: LaunchSettings inside ASP.NET CORE Project

To compile and push this to production we opted for creating a Dockerfile for our project that takes the project into an docker image for our docker-compose to run, as in fig. 3. we later pushed this Dockerfile into a DockerHub repository that we can pull from in our ‘docker-compose.yml‘ file.

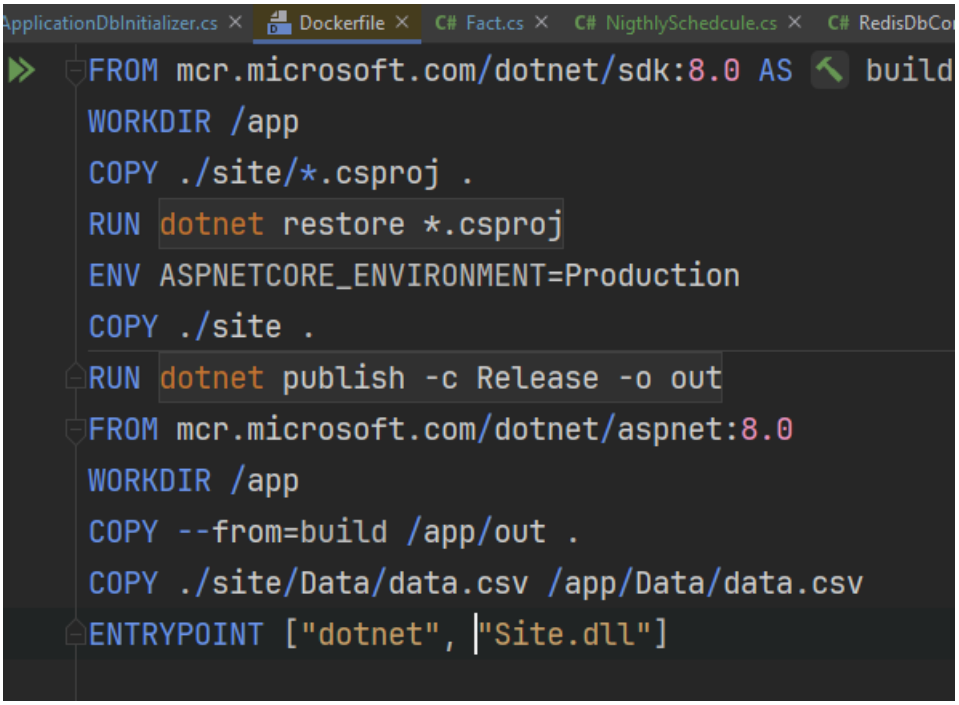


Figure 3: DockerFile

3.2 Database Initialisation

With the way we are introducing the databases to the website using ASP .net, we have decided to use docker images to run and host the website that connects to the databases. This involves setting up an pipeline where we first build the project, then run the docker compose that starts both the databases as well as the server. Furthermore after the server and databases are ready to handle incoming data we connect them to each other through the server.

3.2.1 Docker compile

To run the databases and the website as previously mentioned we intended to use docker images. To run said docker images we needed to configure a docker compose file that we could use to docker compose. The following figure 4 is the configuration of the web server with the databases.

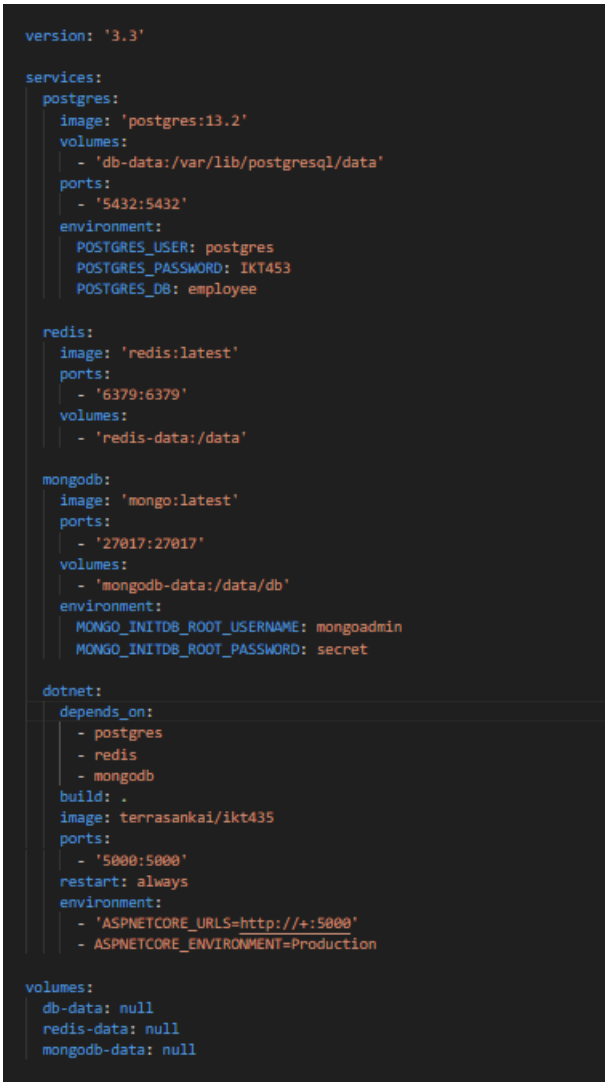


Figure 4: Docker Compose file.

As seen in the previous figure, the server, along side the databases, are now running and ready to handle data, as shown in figure 5.

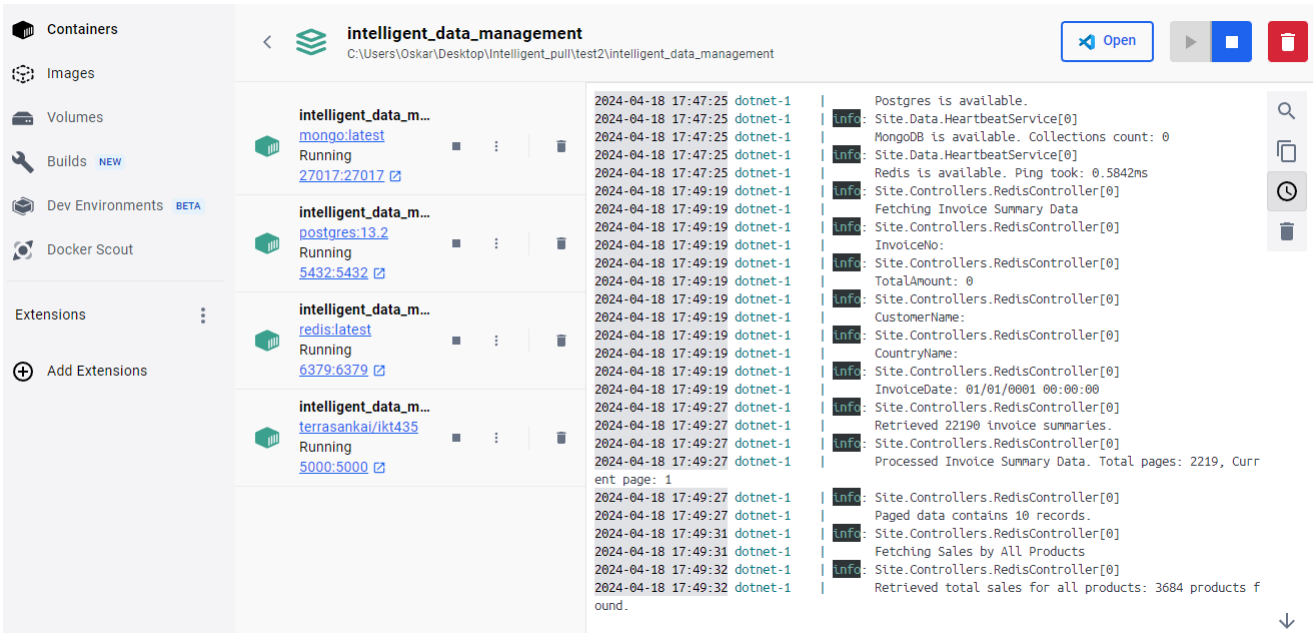


Figure 5: Docker running server and databases

3.3 Database Connection

Connecting to each database was done in similar manners to each other, but had slight differences based on the compatible clients each database needed.

3.3.1 PostgreSQL connection

In our project, we configured PostgreSQL as our relational database to manage structured data with complex relationships. To connect to PostgreSQL from our ASP.NET Core application, we utilized Entity Framework Core with the Npgsql provider. We configured the connection string in the "appsettings.json" file. Which allows us to dynamically adjust our database connection details without modifying the code. In the "Startup.cs" file, we added the ApplicationDbContext to the service container and configured it to use Npgsql with the connection string fetched from the configuration. We decided to name the PostgreSQL context ApplicationDbContext, instead of the more logical PostgreSQLDbContext because we originally treated it as the primary database of the application, before adding on further. This setup allows us to leverage Entity Framework Core's ORM capabilities for data access and manipulation. The figure 6 shows how the connection is established in the "Startup.cs" file, while the figure 7 shows how the configuration is done in the "appsettings.json" file.

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseNpgsql(
        Configuration.GetConnectionString("DefaultConnection")));
```

Figure 6: PostgreSQL connection in Startup file

```
"ConnectionStrings": {
  "DefaultConnection": "Host=postgres;Port=5432;Username=postgres;Password=IKT453;Database=postgres;",
  "MongoDbConnection": "mongodb://mongoadmin:secret@mongodb:27017",
  "RedisConnection": "redis:6379,abortConnect=false"
},
```

Figure 7: Connection string in appsetting.json

3.3.2 MongoDB connection

Similar to the PostgreSQL, we defined the MongoDB service in our Docker Compose configuration, ensuring that the MongoDB instance is accessible to our application.

In the Startup.cs file, we established a connection to MongoDB by configuring the IMongoClient instance as a singleton service. We retrieved the MongoDB connection string from appsettings.json, ensuring our application can communicate with MongoDB to perform CRUD operations on documents. The singleton service we use is to add a dependency injection to our project. The figure 8 shows how the connection is established in the "Startup.cs" file, while the figure 7 shows how the configuration is done in the "appsettings.json" file.

```
services.AddSingleton<IMongoClient>(new MongoClient(Configuration.GetConnectionString("MongoDbConnection")));
```

Figure 8: MongoDB connection in Startup file

3.3.3 Redis connection

We also integrated Redis into our ASP.NET Core application by registering the IConnectionMultiplexer as a singleton service within the Startup.cs file. This connection multiplexer is configured to connect to Redis using the connection string defined in appsettings.json. Furthermore, we implemented a RedisInitializerService as a hosted service to perform any necessary initialization tasks for Redis, such as pre-loading data or setting up default cache values. This RedisInitializerService was beneficial, but not necessary as ASP.NET does provides a flexible configuration system to manage settings like dataset paths and a generalized approach to service initialization, however our choice to create a dedicated service for Redis was because we wanted to test the possible benefits and easier management.

The figure 9 shows how the connection is established in the "Startup.cs" file, while the figure 7 shows how the configuration is done in the "appsettings.json" file.

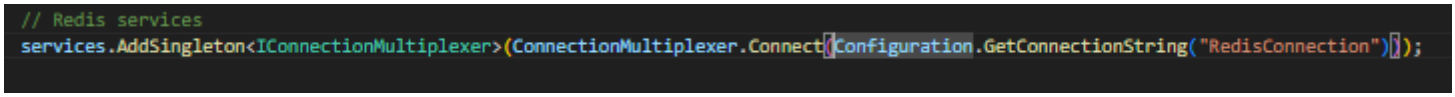


Figure 9: Redis connection in Startup file

3.3.4 Heartbeat Service

As we now are connected to the databases with our website, we now want to establish periodical knowledge of if we are connected to the server or not. This is why we developed a HeartbeatService as part of our ASP.NET Core application to periodically check the availability of our application's dependencies. This include necessary entities such as databases and other services. This service implements the IHostedService interface, which makes it able to run in the background throughout the application's lifetime and we use IDisposable for proper cleanup of resources after.

Upon service startup, signaled by our StartAsync method, we log that the Heartbeat Service is running and initialize a Timer. This timer triggers the DoWork method at specified intervals—in our case, every 5 minutes—though this interval can be adjusted according to our needs.

The DoWork method is designed to be non-blocking and runs its checks in an asynchronous task. Within this method we invoke separate asynchronous tasks to check the availability of PostgreSQL, MongoDB, and Redis. These checks are performed using the respective client libraries for each service.

The figure 10,11 ,12 and 13 shows the implementation of the heartbeat services.

```
namespace Site.Data
{


    public class HeartbeatService : IHostedService, IDisposable
    {
        private Timer _timer;
        private readonly ILogger<HeartbeatService> _logger;
        private readonly IServiceProvider _serviceProvider;

        public HeartbeatService(ILogger<HeartbeatService> logger, IServiceProvider serviceProvider)
        {
            _logger = logger;
            _serviceProvider = serviceProvider;
        }

        public Task StartAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation("Heartbeat Service running.");

            _timer = new Timer(DoWork, null, TimeSpan.Zero,
                TimeSpan.FromMinutes(5));

            return Task.CompletedTask;
        }

        private void DoWork(object state)
        {
            _ = Task.Run(async () =>
            {
                await CheckDatabaseAvailabilityAsync(_serviceProvider);
                await CheckMongoDbAvailabilityAsync(_serviceProvider);
                await CheckRedisAvailabilityAsync(_serviceProvider);

            });
        }
```

Figure 10: Startup of Heartbeat Service

```
private async Task CheckDatabaseAvailabilityAsync(IServiceProvider services)
{
    using (var scope = services.CreateScope())
    {
        var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
        try
        {
            // Using the asynchronous method to check the database connection
            var canConnect = await dbContext.Database.CanConnectAsync();
            if (canConnect)
            {
                _logger.LogInformation("Postgres is available.");
            }
            else
            {
                _logger.LogWarning("Postgres is unavailable.");
            }
        }
        catch (Exception ex)
        {
            _logger.LogError($"Postgres check failed: {ex.Message}");

        }
    }
}
```

Figure 11: Heartbeat Service PostgreSQL

```
private async Task CheckRedisAvailabilityAsync(IServiceProvider services)
{
    using (var scope = services.CreateScope())
    {
        var connectionMultiplexer = scope.ServiceProvider.GetRequiredService<IConnectionMultiplexer>();
        try
        {
            // Perform a simple PING to check if Redis is available
            var db = connectionMultiplexer.GetDatabase();
            var pingResult = await db.PingAsync();
            _logger.LogInformation($"Redis is available. Ping took: {pingResult.TotalMilliseconds}ms");
        }
        catch (Exception ex)
        {
            _logger.LogError($"Redis check failed: {ex.Message}");

        }
    }
}
```

Figure 12: Heartbeat Service Redis

```
private async Task CheckMongoDbAvailabilityAsync(IServiceProvider services)
{
    using (var scope = services.CreateScope())
    {
        var mongoClient = scope.ServiceProvider.GetRequiredService<IMongoClient>();
        try
        {
            var database = mongoClient.GetDatabase("MyAppDatabase");
            // Await the asynchronous operation and then call ToListAsync on the result
            var collectionsCursor = await database.ListCollectionNamesAsync();
            var collections = await collectionsCursor.ToListAsync();
            _logger.LogInformation($"MongoDB is available. Collections count: {collections.Count}");
        }
        catch (Exception ex)
        {
            _logger.LogError($"MongoDB check failed: {ex.Message}");

        }
    }
}



public Task StopAsync(CancellationToken stoppingToken)
{
    _logger.LogInformation("Heartbeat Service is stopping.");

    _timer?.Change(Timeout.Infinite, 0);

    return Task.CompletedTask;
}

public void Dispose()
{
    _timer?.Dispose();
}
```

Figure 13: Heartbeat Service MongoDB and clean up

3.4 ASP .NET MVC

In our project we follow a version of the ASP .NET MVC pattern or Model View Controller pattern, This involves storing the structure for the database in the Model, showing the pages and results of queries in the View and receiving and doing the queries in the Controller. We have a slight different iteration of this which is with the addition of Data. The Data is where the Model will be populated on startup.

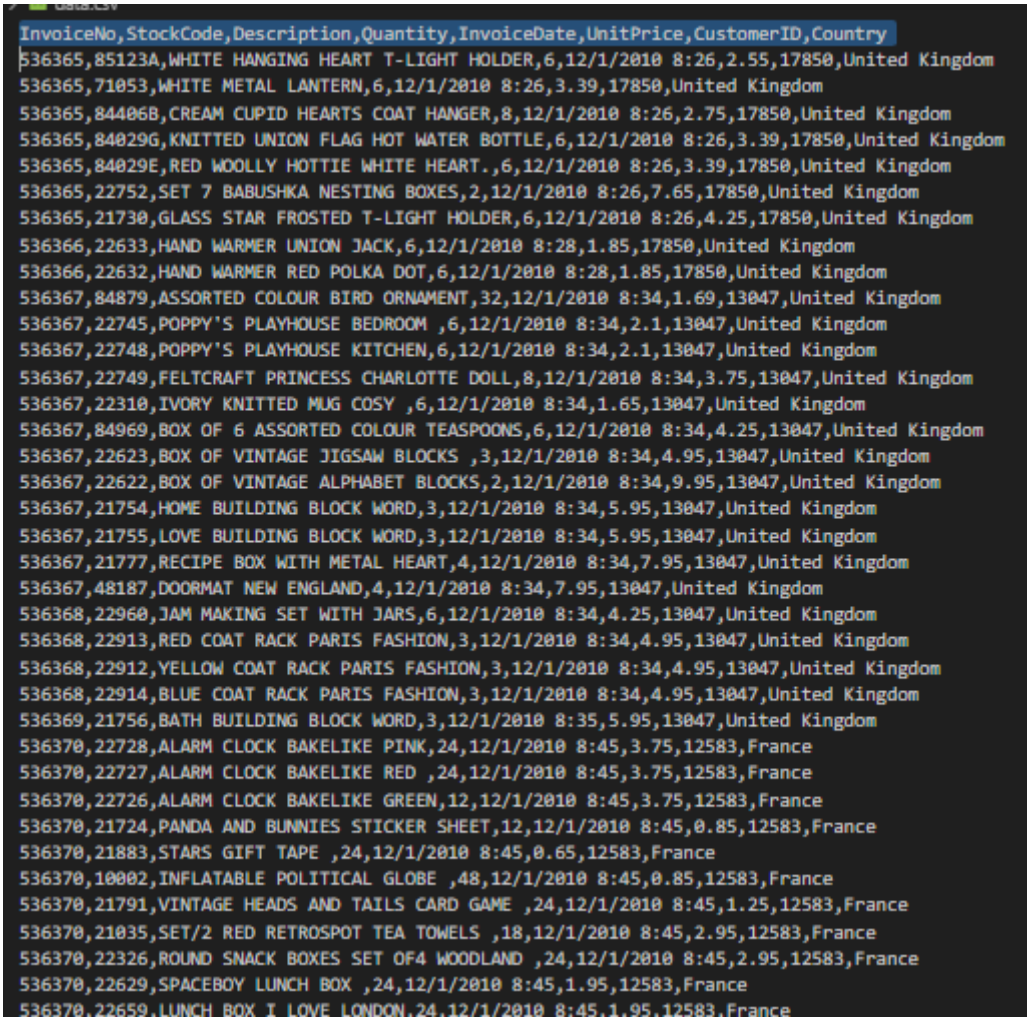
3.5 Initialization

Following the connection to the databases, we initialize the pre-aggregated data from a csv E-commercial dataset. Each database is a little different in the ways they are storing data and is why there is different initialization for each database. As PostgreSQL follows a relational data-structure we have chosen to add a star schema like structure to initialize our data. However this way of structuring data, was discovered by us to be more difficult to do with the non-relational databases such as MongoDB and Redis. This is why we have decided to use a similar structure but with some differentiation. Some of the differences involves pre-aggregation of queries in Redis since the databases does not use an SQL-like querying approach, and possibility of join documentation in MongoDB. The reason we pre-aggregated data in Redis is to attempt a faster query in real time since we have decided to use pre-defined queries as demonstrations in our project. We could have chosen to do computations in real life time with Redis but opted for trying out if it would increase the speed of the query, in which it does.

3.5.1 Data Stored In System

When Initializing our system we, as previously mentioned, uses an E-commercial dataset. This dataset contains the information about invoices and the trades done to the company. It involves an Invoice Number that indicates the unique purchase. It contains the stock code and price of the product, as well as the description of what the product is. Following this, it also contains the date of the purchase, how much was purchased, which country it was bought from, and the customer that bought it.

An example of how the csv file looks like is shown in figure 14.



```
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,12/1/2010 8:26,2.55,17850,United Kingdom
536365,71053,WHITE METAL LANTERN,6,12/1/2010 8:26,3.39,17850,United Kingdom
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,12/1/2010 8:26,2.75,17850,United Kingdom
536365,84029G,KNITTED UNION FLAG HOT WATER BOTTLE,6,12/1/2010 8:26,3.39,17850,United Kingdom
536365,84029E,RED WOOLLY HOTTIE WHITE HEART.,6,12/1/2010 8:26,3.39,17850,United Kingdom
536365,22752,SET 7 BABUSHKA NESTING BOXES,2,12/1/2010 8:26,7.65,17850,United Kingdom
536365,21730,GLASS STAR FROSTED T-LIGHT HOLDER,6,12/1/2010 8:26,4.25,17850,United Kingdom
536366,22633,HAND WARMER UNION JACK,6,12/1/2010 8:28,1.85,17850,United Kingdom
536366,22632,HAND WARMER RED POLKA DOT,6,12/1/2010 8:28,1.85,17850,United Kingdom
536367,84879,ASSORTED COLOUR BIRD ORNAMENT,32,12/1/2010 8:34,1.69,13047,United Kingdom
536367,22745,POPPY'S PLAYHOUSE BEDROOM ,6,12/1/2010 8:34,2.1,13047,United Kingdom
536367,22748,POPPY'S PLAYHOUSE KITCHEN,6,12/1/2010 8:34,2.1,13047,United Kingdom
536367,22749,FELTCRAFT PRINCESS CHARLOTTE DOLL,8,12/1/2010 8:34,3.75,13047,United Kingdom
536367,22310,IVORY KNITTED MUG COSY ,6,12/1/2010 8:34,1.65,13047,United Kingdom
536367,84969,BOX OF 6 ASSORTED COLOUR TEASPOONS,6,12/1/2010 8:34,4.25,13047,United Kingdom
536367,22623,BOX OF VINTAGE JIGSAW BLOCKS ,3,12/1/2010 8:34,4.95,13047,United Kingdom
536367,22622,BOX OF VINTAGE ALPHABET BLOCKS,2,12/1/2010 8:34,9.95,13047,United Kingdom
536367,21754,HOME BUILDING BLOCK WORD,3,12/1/2010 8:34,5.95,13047,United Kingdom
536367,21755,LOVE BUILDING BLOCK WORD,3,12/1/2010 8:34,5.95,13047,United Kingdom
536367,21777,RECIPE BOX WITH METAL HEART,4,12/1/2010 8:34,7.95,13047,United Kingdom
536367,48187,DOORMAT NEW ENGLAND,4,12/1/2010 8:34,7.95,13047,United Kingdom
536368,22960,JAM MAKING SET WITH JARS,6,12/1/2010 8:34,4.25,13047,United Kingdom
536368,22913,RED COAT RACK PARIS FASHION,3,12/1/2010 8:34,4.95,13047,United Kingdom
536368,22912,YELLOW COAT RACK PARIS FASHION,3,12/1/2010 8:34,4.95,13047,United Kingdom
536368,22914,BLUE COAT RACK PARIS FASHION,3,12/1/2010 8:34,4.95,13047,United Kingdom
536369,21756,BATH BUILDING BLOCK WORD,3,12/1/2010 8:35,5.95,13047,United Kingdom
536370,22728,ALARM CLOCK BAKELIKE PINK,24,12/1/2010 8:45,3.75,12583,France
536370,22727,ALARM CLOCK BAKELIKE RED ,24,12/1/2010 8:45,3.75,12583,France
536370,22726,ALARM CLOCK BAKELIKE GREEN,12,12/1/2010 8:45,3.75,12583,France
536370,21724,PANDA AND BUNNIES STICKER SHEET,12,12/1/2010 8:45,0.85,12583,France
536370,21883,STARS GIFT TAPE ,24,12/1/2010 8:45,0.65,12583,France
536370,10002,INFLATABLE POLITICAL GLOBE ,48,12/1/2010 8:45,0.85,12583,France
536370,21791,VINTAGE HEADS AND TAILS CARD GAME ,24,12/1/2010 8:45,1.25,12583,France
536370,21035,SET/2 RED RETROSPOT TEA TOWELS ,18,12/1/2010 8:45,2.95,12583,France
536370,22326,ROUND SNACK BOXES SET OF4 WOODLAND ,24,12/1/2010 8:45,2.95,12583,France
536370,22629,SPACEBOY LUNCH BOX ,24,12/1/2010 8:45,1.95,12583,France
536370,22659,LUNCH BOX I LOVE LONDON,24,12/1/2010 8:45,1.95,12583,France
```

Figure 14: Example of how the dataset look like

From this information provided by the dataset, we decided to add another attribute to database which was the total value of each transaction, which was done by multiplying the amount purchased with the cost of the product.

3.5.2 Star Schema

Our data is arranged using a star schema in a sales data warehouse, with dimension tables surrounding the center fact table. This schema is useful for analytical applications in a data warehouse setting since it is optimized for querying and reporting instead of transaction processing.

Quantitative measurements of business transactions, such as quantities of products sold, unit prices, and calculated total prices, are included in the fact table located at the center of the star schema. Moreover, it has keys that connect to other dimension tables. These keys are essential because they define the connection between the descriptive qualities kept in the dimension databases and the transaction data.

Dimension tables in the star schema provide context to the data captured in the fact table. These tables include:

- Date Dimension: Stores comprehensive date information such as year, month, and day, facilitating time-based analysis and reporting.
- Product Dimension: Contains details about products such as descriptions and stock codes, which are useful for product-specific analysis.
- Customer Dimension: Includes customer-specific information, which helps in analyzing purchasing patterns and customer behavior.
- Country Dimension: Captures geographical data, allowing for regional sales performance analysis.

Complex queries will get data efficiently because of the star scheme's simple architecture, which connects each dimension table to a single fact table. The process of aggregating data across several dimensions—for example, total sales per product or long-term sales trend analysis—is made simpler by this structure. Large data volumes are handled with this architecture in a PostgreSQL data warehouse, which also offers notable speed gains for tasks like reporting and querying that are essential to business intelligence operations.
For the Data Definition Language, see Appendix B.

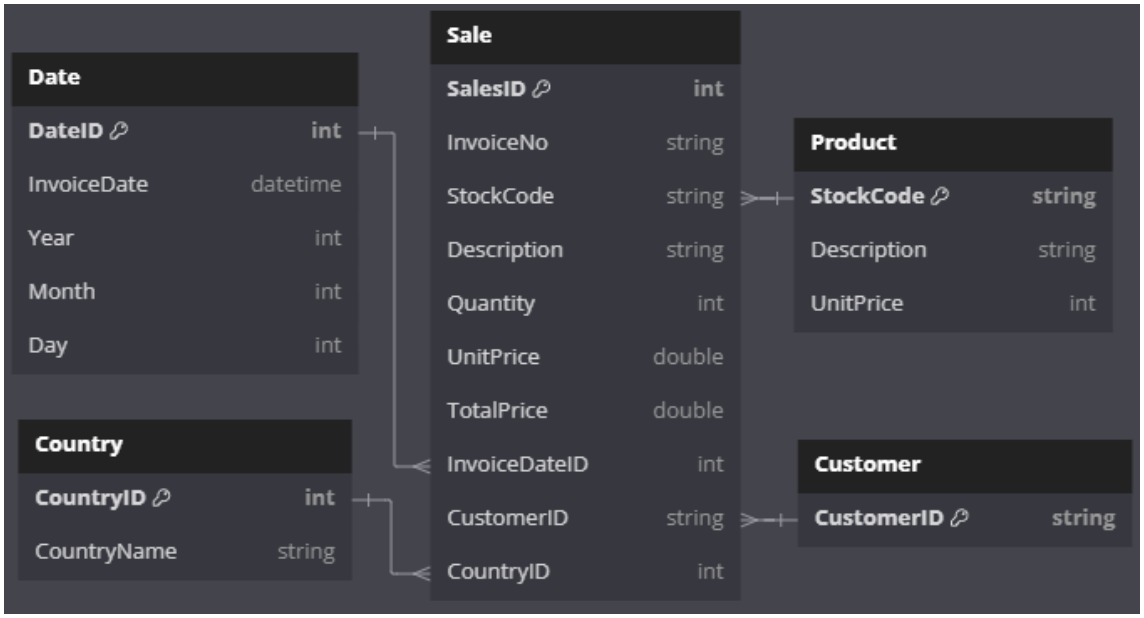


Figure 15: Star Schema illustration

3.6 PostgreSQL Overview

3.6.1 Data Model and Structure

In the initial phase of integrating PostgreSQL into our application, we crafted a data model that reflected the entities identified within the CSV file, this included Sales, Products, Customers, Countries, and Dates. Each of the entities was represented as a class in our application, with properties directly mapping to the corresponding columns in the CSV file. Contradicting to our Redis database we did not pre-aggregate the queries, but rather choose to do the SQL transactions in real time.

3.6.2 PostgreSQL Database Initialization

With our data model in place, the next step involved initializing the PostgreSQL database, a process that entailed importing and organizing the data from the CSV file into our database schema. We stored the data in tables similar to the star schema shown previously. By doing so we could utilized PostgreSQL's relational model to maintain a structured and efficient storage mechanism, which was a key features for data retrieval and query performance. The figure 16 shows an example of how the addition of data to records is done.

```
private static void AddDateRecords(ApplicationDbContext db, HashSet<DateTime> dates)
{
    var existingDates = db.Dates.Select(d => d.InvoiceDate).ToHashSet();
    var newDates = dates.Except(existingDates).Select(d => new Date(d));
    db.Dates.AddRange(newDates);
}
```

Figure 16: PostgreSQL adding of records to tables

3.6.3 Querying PostgreSQL

Querying in our PostgreSQL integration emphasized the utilization of SQL for both data retrieval and the execution of queries. This approach capitalized on PostgreSQL's efficient data processing capabilities and its ability to execute complex SQL queries with high performance.

Some of the queries we made was for example Total Sales by Country which was to retrieve aggregated sales data by country, we executed SQL queries that utilized the GROUP BY clause, enabling the efficient aggregation of sales totals directly within the database. This method exemplified the power of SQL in performing complex data aggregations in a relational database environment. Another query we did was the Sales Trends. Where our queries took advantage of PostgreSQL's date and time functions alongside aggregation clauses. By organizing sales data by specific time intervals (e.g., monthly sales trends). Another query we did was the Customer Lifetime Value which involved calculating the customer lifetime value. This aggregation was performed through SQL queries that grouped sales data by customer ID. The figure 17 shows an example of how the SQL querying is done in PostgreSQL.

```
// Fetch initial data
var data = await _dbContext.Sales
    .Select(x => new { x.CustomerID, x.TotalPrice })
    .ToListAsync();

// Perform grouping and aggregation in-memory
var groupedData = data
    .GroupBy(f => f.CustomerID)
    .Select(group => new CustomerLifetimeValueViewModel
    {
        CustomerID = group.Key,
        LifetimeValue = group.Sum(g => g.TotalPrice)
    })
    .AsQueryable();

// Apply dynamic sorting
var sortedData = groupedData.OrderByDynamic(sortField, sortOrder);
```

Figure 17: Example of Query done in PostgreSQL

3.7 MongoDB Overview

3.7.1 Data Model and Structure

The MongoDB database began with a design of our data model, reflecting the entities identified in the CSV file, such as Sales, Products, Dates, Customers, and Countries. Each entity was encapsulated within a class, mirroring the columns in the CSV. This allows for a direct mapping between the data file and our MongoDB documents. This approach facilitated a structured yet flexible schema design, characteristic of MongoDB' s document-oriented nature.

3.7.2 MongoDB Database Initialization

Following the data preparation, we started on initializing our MongoDB database. This process involved serializing pre-aggregated data into BSON documents and storing them within respective collections in MongoDB, which was creatively named "MyAppDatabase". Meaning an early step in this integration was the instantiation of collections such as MongoSale, which directly mapped to our sales data, enabling a easier interaction with our data stored in MongoDB. The figure 18 shows an example on how the records is added to the documents.

```
if (bulkCustomers.Any())
{
    customersCollection.BulkWrite(bulkCustomers);
    Console.WriteLine("Customers bulk inserted. Sample Customer ID: {0}", ((InsertOneModel<MongoCustomer>)bulkCustomers.Fir
}
```

Figure 18: MongoDB adding of records to Documents

3.7.3 Querying MongoDB

Querying our MongoDB database was a dynamic process, reliant on MongoDB's aggregation framework. The aggregation operations made us able to perform data manipulations, such as grouping sales data by country or calculating total sales over specific periods. MongoDB's query capabilities were used in scenarios such as:

Total Sales by Country, where we utilized MongoDB's aggregation pipeline, we grouped sales documents by country and calculated the total sales for each. This gave us the amount each country had used at total.

We looked at Sales Trends where we by aggregating sales data based on the invoice date, we were able to identify sales trends over time. MongoDB's ability to handle date-time operations within the aggregation framework proved valuable in querying sales patterns.

Another query we did was the Customer Lifetime Value which with MongoDB's aggregation capabilities also facilitated the calculation of customers lifetime values. By grouping sales data by customer ID and aggregating the total sales, we got the information about the customers value over time.

These query implementations highlighted the importance of effective indexing and sorting to optimize query performance, as we had observed a stark difference in performance when indexations and sorting was done properly. The figure 19 shows an example of querying in MongoDB.

```
// Performing the aggregation
var aggregation = _salesCollection.Aggregate()
    .Group(sale => sale.CountryName, g => new
    {
        CountryName = g.Key,
        TotalSales = g.Sum(x => x.Quantity * x.UnitPrice) // Calculating total sales
    })
    .Project(sale => new TotalSalesByCountryViewModel
    {
        Country = sale.CountryName,
        TotalSales = sale.TotalSales
    })
    .Sort(sortDefinition)
    .Skip((pageNumber - 1) * pageSize)
    .Limit(pageSize);
```

Figure 19: Example of Query done in MongoDB

3.8 Redis Overview

3.8.1 Data Model and Structure

Initially we defined a clear data model based on the entities present in the CSV file, this included Sales, Products, Dates, Customers, and Countries. Each entity is represented as a class in our program, with properties reflecting the data columns in the CSV file. However after some more implementation and understanding of the nature of how Redis stores and retrieves data, we decided to implement pre-aggregated queries to increase performance, and make it more suitable to the methods often used by No-SQL databases.

3.8.2 Redis Database Initialization

Following the data preparation, our focus shifted towards initializing the Redis database. This entailed the serialization of pre-aggregated data, which was then stored within Redis utilizing distinct key patterns tailored for efficient data retrieval. Our strategy involved computing aggregates such as total sales by country and customer lifetime values upfront. These aggregates were serialized into JSON format and stored under specifically designed keys in Redis, such as `totalSalesByCountry:{countryName}` for the total sales per country, showcasing the adaptability and performance efficiency of Redis for storing aggregated data. The table 20 shows an example of how records of pre-aggregated query are added to Redis as key value pairs. Including this, all the additions of data was added to the database in one batch instead of separately to decrease start up time.

```
// Adding total sales per product
foreach (var kvp in totalSalesByProduct)
{
    var productSalesData = new ProductSalesViewModel
    {
        ProductStockCode = kvp.Key,
        Quantity = kvp.Value.Quantity,
        UnitPrice = kvp.Value.UnitPrice,
        TotalAmount = kvp.Value.TotalAmount
    };
    var key = $"productTotalSales:{kvp.Key}";
    var value = JsonConvert.SerializeObject(productSalesData);
    tasks.Add(batch.StringSetAsync(key, value));
}
```

Figure 20: Redis adding of records to key value pairs

3.8.3 Querying Redis

In our Redis integration, our querying revolves around the utilization of pre-aggregated data, which is stored as key-value pairs. This approach shows some of the Redis methodology, where data retrieval is orchestrated through keys made to match the query intent.

A example of this strategy was our Total Sales by Country query which is the retrieval of total sales figures for countries. By invoking keys that conform to the totalSalesByCountry:* pattern, our system accesses pre-aggregated totals for each country, showing the fast retrieval method utilised by Redis.

Similarly, for analyzing sales trends, we use keys structured as salesTrend:year-month, a design choice that enables the fetching of sales data, which corresponding to specific time frames. This method showcases the way Redis can support time-series data analysis, allowing for the quick summarization of sales trends without the need for much computation.

Another query we did is the querying of customer lifetime values. By storing these values under keys like customerLifetimeValue:customerId, we ensure that the lifetime value of any given customer can be retrieved in an fast performing time, highlighting the directness of querying in Redis.

The pre-aggregation of data also eliminates the computational overhead associated with real-time data aggregation. The figure 21 shows an example of how the querying in Redis was done.

```
var db = _redis.GetDatabase();
var server = _redis.GetServer(_redis.GetEndPoints().First());
var pattern = "invoiceSummary:*";
var keys = server.Keys(database: db.Database, pattern: pattern);

var invoiceSummaries = new List<Site.ViewModels.InvoiceSummaryViewModel>();

foreach (var key in keys)
{
    var value = await db.StringGetAsync(key);
    if (!value.IsNullOrEmpty)
    {
        var summary = JsonConvert.DeserializeObject<Site.ViewModels.InvoiceSummaryViewModel>(value);
        invoiceSummaries.Add(summary);
    }
}

// Log the count of retrieved summaries
_logger.LogInformation($"Retrieved {invoiceSummaries.Count} invoice summaries.");

// Dynamic ordering
IEnumerable<Site.ViewModels.InvoiceSummaryViewModel> query = orderBy switch
{
    "TotalAmount" => invoiceSummaries.OrderByDescending(i => i.TotalAmount),
    "CustomerName" => invoiceSummaries.OrderBy(i => i.CustomerName),
    _ => invoiceSummaries.OrderBy(i => i.InvoiceNo),
};
```

Figure 21: Example of Query done in Redis

3.9 Queries To Be Done On The System

After initializing, we decided on some database queries which would demonstrate applications on the E-commerce dataset. Here are some of the queries we settled on:

- **Total Sales by Country:** We can calculate the total sales figures for each country based on sales records. The total sales amount for each country can be computed as:

$$\text{Total Sales}_{\text{Country}} = \sum_{\text{Sales}} (\text{Quantity} \times \text{Unit Price})$$

- **Total Sales by Product:** find the total units sold, total sales revenue, and latest unit price for each product:

$$\text{Total Units Sold}_{\text{Product}} = \sum_{\text{Sales}} \text{Quantity}$$

$$\text{Total Sales Revenue}_{\text{Product}} = \sum_{\text{Sales}} (\text{Quantity} \times \text{Unit Price})$$

- **Invoice Summaries:** Generate an invoice summary for each invoice containing the total amount due, customer name (or ID), country name, and invoice date.

– **Customer Lifetime Values (CLV):** Add together all of the sales amounts linked to each customer's ID to determine their lifetime value:

$$\text{CLV}_{\text{Customer}} = \sum_{\text{Sales}} (\text{Quantity} \times \text{Unit Price})$$

– **Sales Trends:** By combining monthly and annual sales data, we can track patterns over time and analyze sales trends.

The following three sub chapters will contain some more detailed explanation of how we have queried in the three implemented databases.

3.9.1 PostgreSQL Querying

To use PostgresSQL SQL's in our system we build SQL queries across several PostgresController endpoints that are specific to the certain data retrieval and manipulation operations. As an example, when we want to ensure all relevant data is retrieved in a single query, we use the Include method to fetch sales data in from both Sales and its connected table Country. Total sales is then calculated by aggregating sales data by nation, using SQL's GROUP BY. This SQL procedure reduces processing overhead and data transfer by summarizing data at the database level.

Similar to this, we aggregate sales data by product in SalesByProduct by using SQL's GROUP BY clause to determine quantities and total prices. We minimize the amount of data transported to the application layer and maximize performance by aggregating data directly within the database.

We compile invoice data in InvoiceSummary by adding invoice dates, client information, and country data, from our table Sales, and its connected tables Customer and Dates, and Country. We may show the invoice data by combining data from different tables with SQL's JOIN technique. Furthermore, we use SQL's GROUP BY clause to aggregate customer purchase data and determine each customer's lifetime value in the tables Customer and Sales. Additionally, we use SQL's GROUP BY clause in SalesTrends to examine sales patterns over time. Sales data can be grouped by month and year to help us spot trends and variations in sales performance. This data is retrived from the tables Sales and Dates. We prioritize using SQL's built-in features to aggregate, filter, and sort data inside the database during the process.

3.9.2 MongoDB Querying

In MongoDB, data is organized as collections of documents, each of which is a single record. With the help of these documents, our MongoDbController carries out a range of query operations. We design and execute aggregation pipelines while querying MongoDB using the features provided by the MongoDB.NET driver. A set of procedures called an aggregation pipeline is used to process documents, modify data, and combine the outcomes.

For example, we use the aggregation framework in the TotalSalesByCountry and SalesByProduct methods to group documents according to certain criteria, like country or product, and then compute the aggregated metrics, like total sales. By taking advantage of MongoDB's

effective aggregation engine, we can swiftly handle large masses of data and carry out the actions directly within the database.

We carry out more intricate aggregations in methods like InvoiceSummary and CustomerLifetimeValue, involving steps to calculate metrics like total sales amounts and customer lifetime values. By using MongoDB's extensive aggregation functionalities, these calculations may be effectively and concisely expressed.

Our implementation also includes $skip and $limit phases in the aggregation pipelines to guarantee effective pagination. we may, through this, mitigate the performance concerns that arise from searching huge databases by retrieving data in pieces.

3.9.3 Redis Querying

The main aim of our query process is to grab aggregated data that we've saved under specific keys, where each key shows a different slice of our data.

A query implemented is to pull up the total sales data by country. In Redis, we save this info under keys like totalSalesByCountry:countryName. This setup means each key has the total sales for that country, making it very quick to find out the sales total for any country we need.

We keep tabs on the overall sales of each product in Redis. Using keys like productTotalSales:stockCode, we store these records. Each key holds the total sales info for a specific product, identified by its stock code. To check the total sales for any product, all we need to do is grab the value stored under its corresponding key.

Invoice summaries are another important part of our data, providing a summary of each transaction. We store these summaries in Redis using keys formatted as invoiceSummary:invoiceNo, where each key corresponds to a unique invoice number. To retrieve the summary for a particular invoice, we simply fetch the value associated with its respective key.

We keep a close watch on client lifetime values to better understand their long-term engagement and buying trends. Stored in Redis under keys like customerLifetimeValue:customerId, each key represents the comprehensive value of an individual customer, identifiable by their unique ID. We can retrieve the associated value from Redis to pinpoint a customer's lifetime value.

We monitor trends over time in sales patterns to identify changes and patterns in consumer behavior. Redis organizes this data into keys corresponding to specific months and years, like salesTrend:year-month. We only need to access the value stored under the appropriate key in order to get sales trends for each given time period.

3.10 Front End

In our website the landing page is controlled by the HomeController that navigates the user to the other Controller that is managing the different Databases. When you interact with the system the application ensures that the correct instances is passed to the correct controller via the dependency injection, the system also creates the controllers and views on-demand, meaning that the user have to be navigated to the correct part of the site for it to exist. The view is created with a cshtml file, meaning that the different views are created with C#

and basic html code, with the help of AnchorTagHelper that ASP.NET CORE provides the views can be implemented with the help of .NET-based APIs from Microsoft. Each query is display within a table that is handled from the controller after you click the button on each of the different database section. The button sends a Get Request to the controller, the controller fetches the data from the databases then transforms the database model's data to the desired view-model, then the cshtml file parses the view-model and populates the tables, then the view-models are wrapped into an sortviewmodel this allows view-model members(eg. InvoiceNo, Date, totalsales) to be sorted by either ascending or descending. Since every view is handled this way we only need to specify which controller that each is passed to configure which database we are using. The pagination is also that is further passed to the correct controller.

```
namespace Site.Models
{
    34 usages  lemoi18
    public class SortCriterion
    {
        128 usages
        public string Field { get; set; }
        128 usages
        public string Direction { get; set; } = "asc";
    }

    47 usages  lemoi18
    public class SortViewModel<T>
    {
        46 usages
        public IEnumerable<T> Data { get; set; }
        188 usages
        public List<SortCriterion> SortCriteria { get; set; } = new List<SortCriterion>();
    }

}
```

Figure 22: SortViewModel

3.11 Night schedule

The nightly consists of a service that is initialized by the dependency injector at the start of the program. The life cycle of the class is a hosted service, meaning its running in the background. Its initialized similarly as the heartbeat service only we added a initial delay before the synchronization starts. So as our assumption, PostgresSQL is our main database, while Redis and MongoDB acts as our secondary's. With this in mind we made the nightly schedule be synchronization to the data that's stored in Postgres, meaning that if we pass new data into the Postgres, then MongoDB and Redis will retrieve the new data in the nightly schedule. This is done by retrieving the data in MongoDB and Redis and comparing them with Postgres's data, and then adding the new data to MongoDB or Redis, if any. Additionally we added a Synchronization functionality for the Admin users, this does the same as the Nightly scheduler only it can be manually triggered with an button that only the admin can access.

3.12 Users and Administrators

In our application, we have implemented an authentication and authorization functionality with the ASP.NET Identity Framework. It is specifically designed to handle security protocols and user interactions. The Identity Framework is a package of tools to manage users, passwords, roles, and more. We have opted for two different types of roles, Administrators and users. The roles have varying levels of access and modifiability of the databases. Administrators are granted permissions that allow them to have user-restricted access such as PUT requests and overviews of the users. The users have restricted access limited to read interaction with the databases. This improves the application's security, by only allowing administrators to alter and change the databases. The implementation involved configuring the Identity framework within our ASP.NET MVC application, setting up the necessary database schema for user and role management, and defining the appropriate role-based authorization policies to enforce security measures consistently across the application.

3.13 Examples scenarios of users interact with the system

The figure below shows the Home page of the application. From here the user can access the Postgres-, MongoDB-, and Redis-database. The 'Titel bar' at the top of the page is visible from all the pages in the application.

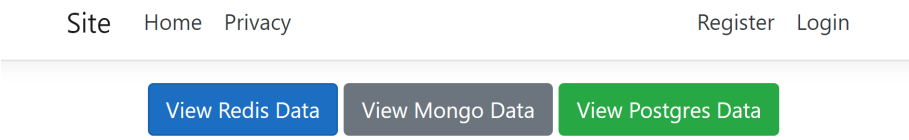


Figure 23: Start Page

The Register button at the right side of the 'Title bar' relocates the user to the 'Create a new user' page. From here the user can create an account with an email and a password.

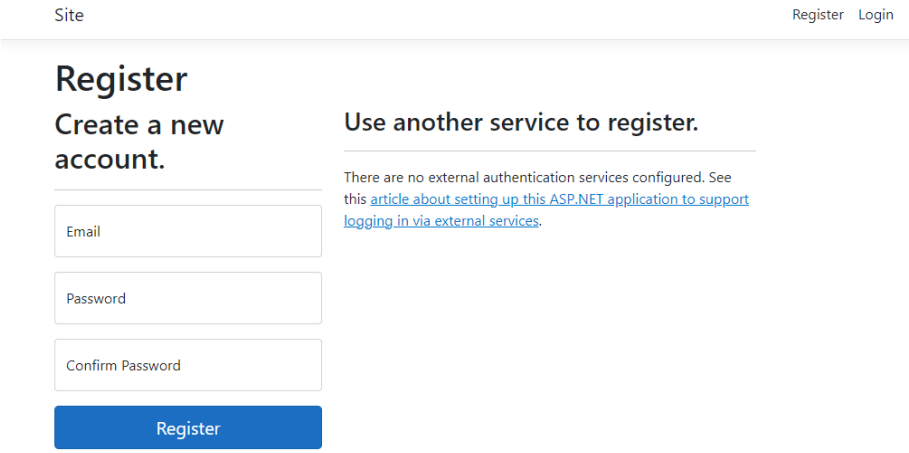


Figure 24: Register Account Page

The Login button at the far right of the 'Title bar' relocates the user to the login page. From here the user can log in to a registered account, reset the password from an email, and opt to have the application remember the user for future log-in.

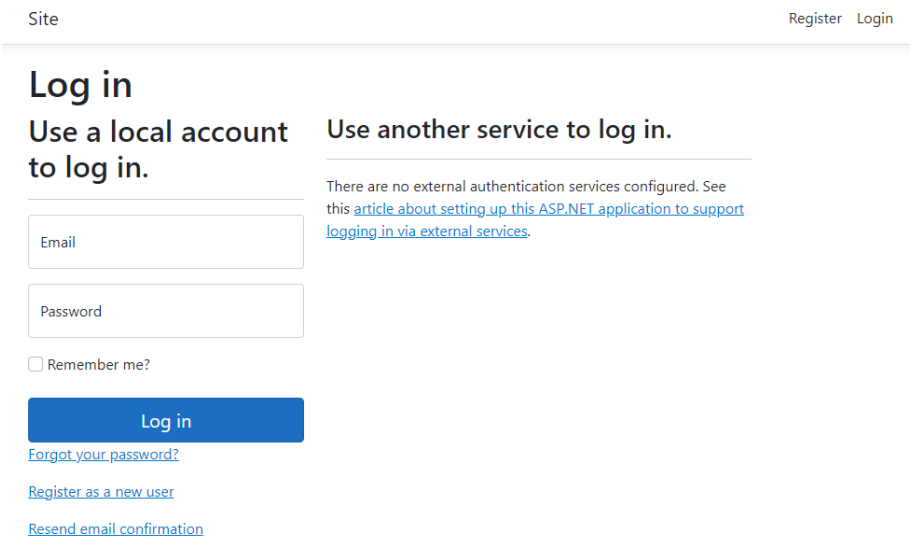


Figure 25: Log in Page

Once the user is logged in, the 'manage account' page is available. From here the user can customize and add information to the account. The user can also reset and change the password to the account, and can also activate two-factor authentication.

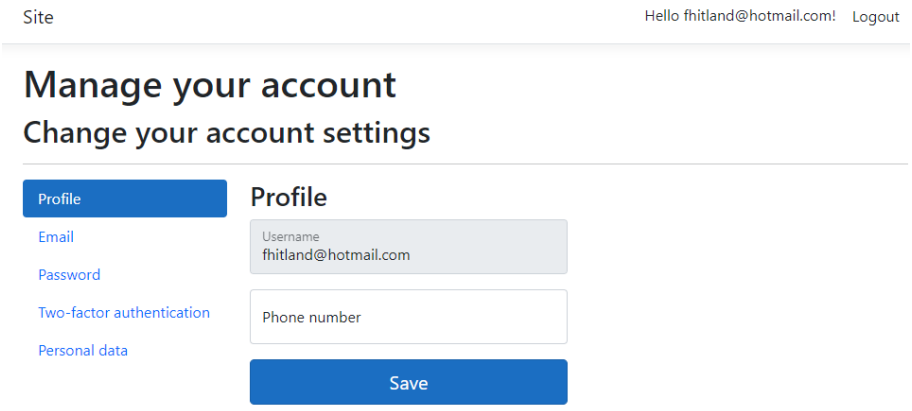


Figure 26: Manage Account

An administrator has an extra button on the home page. The 'Database Sync' relocates the administrator to the 'Add New Sale' page.

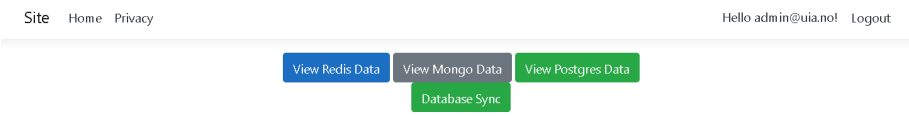


Figure 27: Administrator View of the Home Page

On the 'Add New Sale' page, the administrator can add data to the databases.

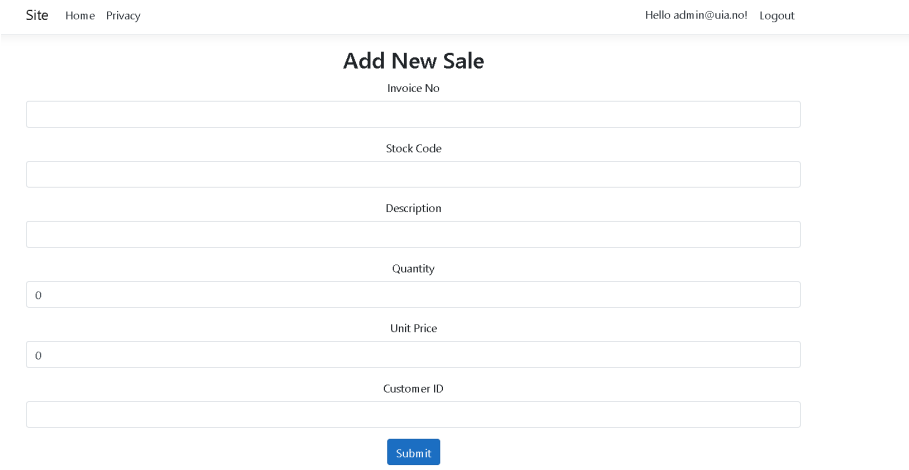


Figure 28: Administrator can add information

When pressing either of the database options, the user will be relocated to the 'Query Options' page. The page has the same visual for each of the databases but is handled differently in the back-end. From here the user has the option to select:

- Invoice Summary
- Sales by Product
- Total Sales by Country
- Customer Lifetime Value
- Sales Trends

Figure 29: Summary Page

If the 'Invoice Summary' button is pressed, the user will be relocated to the 'Invoice Summary' page. This page contains a summary of each invoice, including the total price of the invoice, the customer name/number, the country the invoice and customer is registered to, and the date of the invoice.

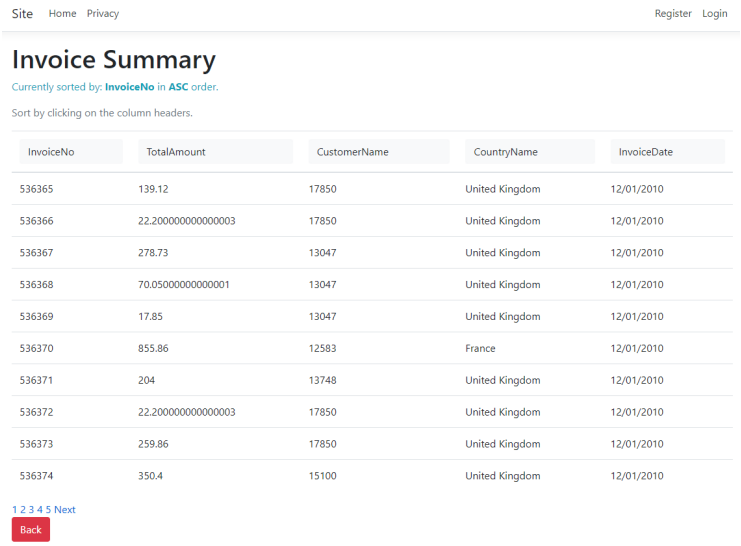


Figure 30: Invoice Summary

If the 'Sales by Product' button is pressed, the user will be relocated to the 'Sales by Product' page. This page contains a summary of sales per product, including the stock code of the product, the quantity sold, the unit price of the product, and the total accumulated money gathered from the sales of the product.

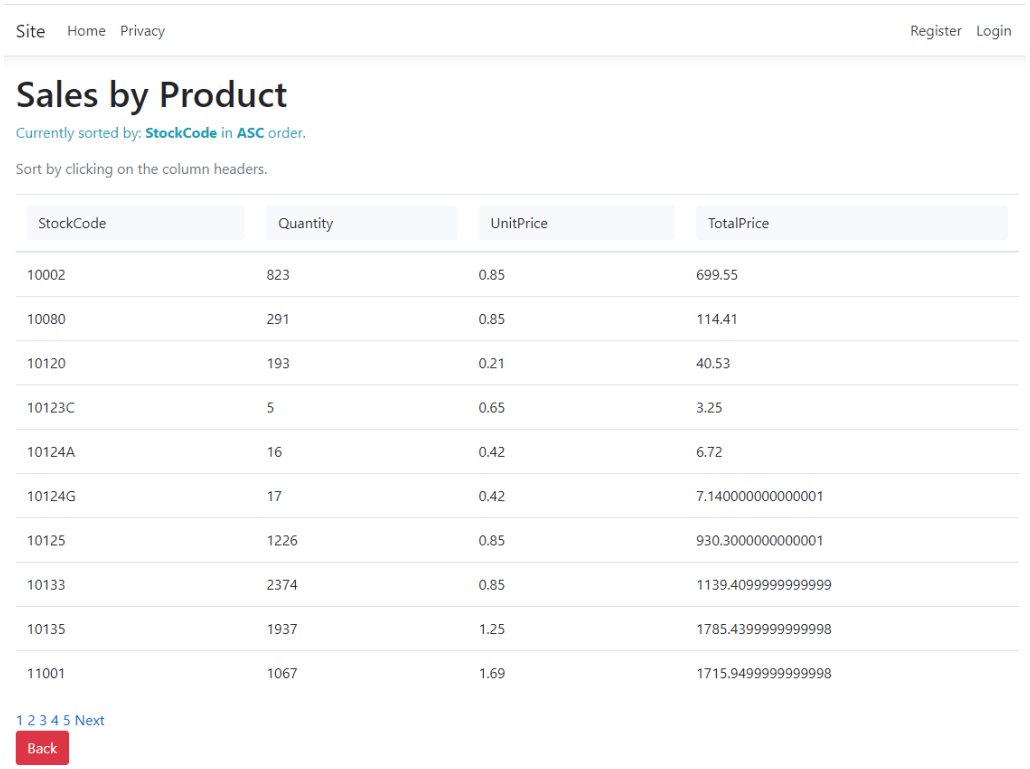


Figure 31: Sales by Product

If the 'Total Sales by Country' button is pressed, the user will be relocated to the 'Total Sales by Country' page. This page contains a summary of the total sales made from each country, including the country name and the total amount generated from all sales in that country.

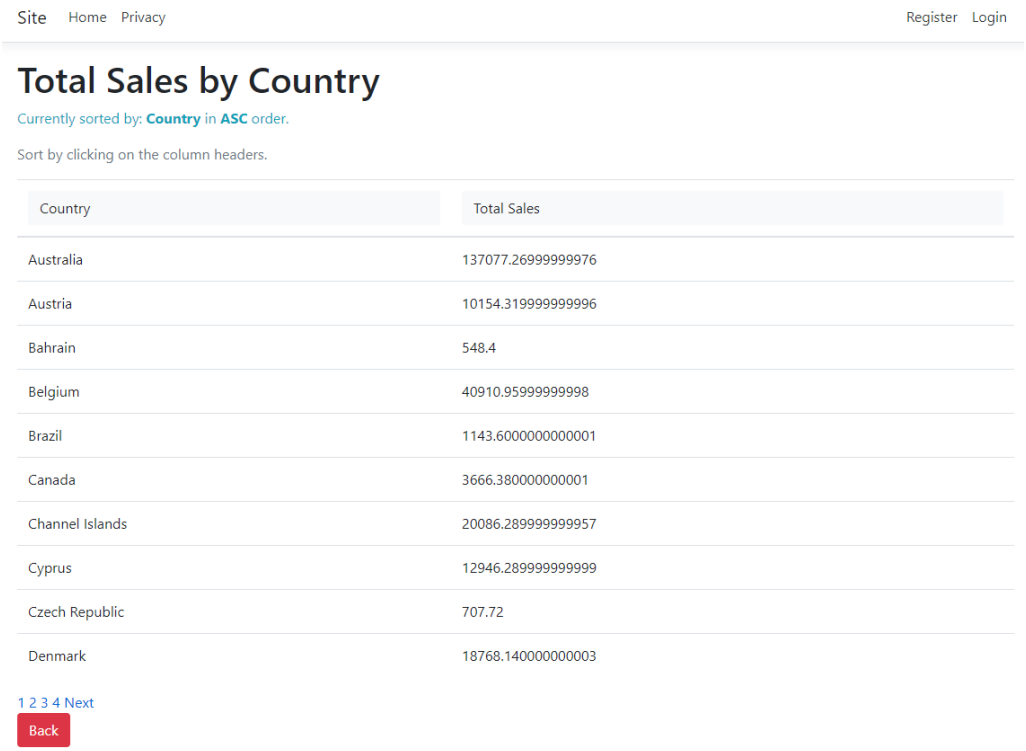


Figure 32: Total Sales by Country

If the 'Customer Lifetime Value' button is pressed, the user will be relocated to the 'Customer Lifetime Value' page. This page contains a summary of how much money a customer has spent during the lifespan of the account.

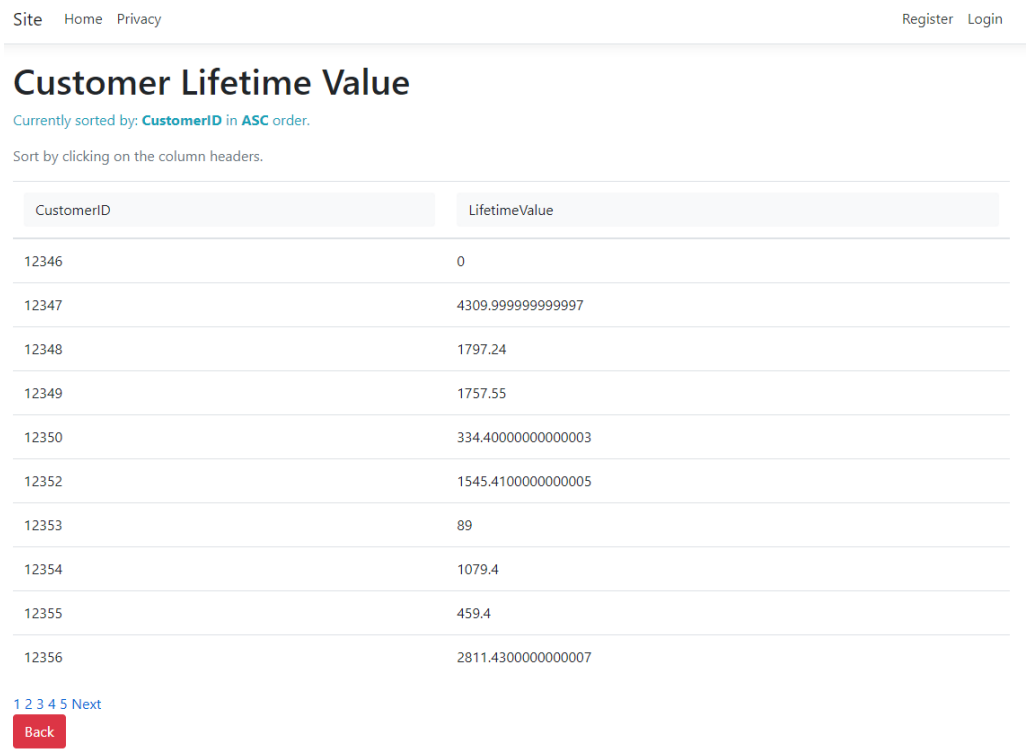


Figure 33: Customer Lifetime Value

If the 'Sales Trend' button is pressed, the user will be relocated to the 'Sales Trend' page. this page contains a summary of how much money the invoices have generated in each month of each year.

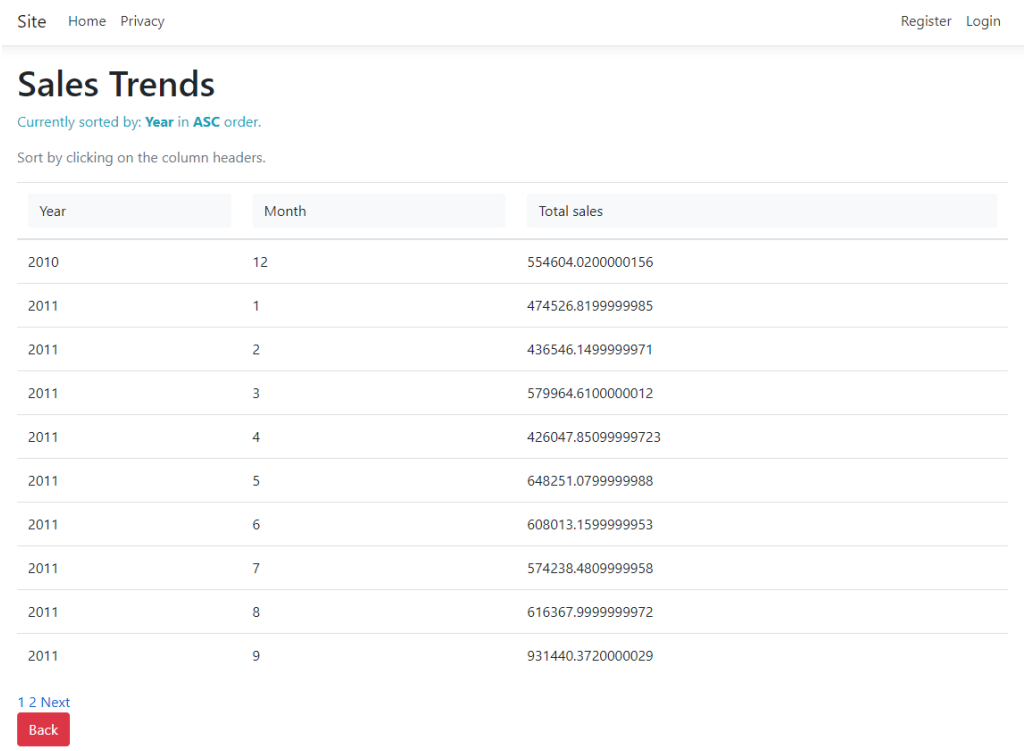


Figure 34: Sales Trends

4 Discussion

4.1 Answer to pre-implementation assumptions

In our experience with implementing a data warehouse that incorporates three distinct types of databases PostgreSQL (relational), MongoDB (document), and Redis (key-value). we had a lot of assumptions prior to the implementation phase. These assumptions stemmed from our understanding and anticipation of how each database type would fit into the architecture of our dataset and choice of framework.

One of our initial assumptions concerned the necessity for pre-aggregation of queries, particularly with the no-SQL databases like MongoDB and Redis. Given their schema-less nature we expected that pre-aggregated data would significantly enhance query performance. We implemented the pre-aggregation of data in our Redis database and our expectations was met as it did in fact make it faster in real-time application at the cost of longer initialisation time. As we needed to Dockerize every time we tested our application we choose to not implement the more complex pre-aggregated queries in our MongoDB and PostgreSQL.

Another assumption revolved around the ease of integrating or transforming of the data models from PostgreSQL, MongoDB, and Redis which we needed to provide a unified view of the data. In hindsight, this proved to be more challenging than anticipated. The differences in data models relational, document-based, and key-value presented an unique integration phase which was quite problematic considering the way data was stored and queried was significantly different from each other.

Regarding PostgreSQL, we assumed that it would excel in handling structured and relational data due to its robust SQL querying capabilities and relational database attributes. We anticipated that PostgreSQL's querying features would be advantageous for complex data relationships and operations, an assumption that was largely validated through our implementation process, however we encountered the problem that it would take a long time to SQL query and join tables during real-time prompting us to reconsider pre-aggregation of the querying next time we implement an similar application.

In terms of performance and scalability, we had assumptions about the no-SQL databases MongoDB and Redis in which we predicted they would offer superior performance metrics, and faster queries. This belief we had partly based on the expectation that pre-aggregation strategies, and direct lookup, as opposed to traditional join and grouping operations found in relational databases like PostgreSQL, would result in a faster query time. While no-SQL databases did show advantages in scalability and performance for certain use cases, the comparison was nuanced and inclusive, and often depended on the specific data operations and contexts.

Lastly, we assumed that integrating these diverse databases within an ASP.NET MVC application would be relatively straightforward, leveraging the Entity Framework for PostgreSQL and utilizing custom or third-party libraries for MongoDB and Redis. We anticipated that the MVC architecture would facilitate a clear separation of concerns, with database interactions neatly encapsulated within the model layer, thereby simplifying the query process through the controller-view connection. While this architectural approach did provide a solid framework for separation of concerns, the integration of different databases introduced complexities that required more time used on the implementation of appropriate abstractions to

manage the different database interactions.

4.2 Post-implementation assumptions

After implementing the three databases, most of our assumptions about the implementation of the system was answered, however we now had some assumptions about the optimisation of the system and how / why some of our expectations where not met.

In our system we looked at the performance of the database queries and discovered that Redis showed the fastest results, followed by MongoDB, then PostgreSQL. This result was expected considering Redis is know for its fast queries based on in memory data storage. Another expected result was that PostgreSQL showed to be slower then both of them. This we think are contributed to the grouping and joining statements used when querying real-time- However we also think the pre-aggregation of Redis data to pre-made querying view-models might have contributed to its relative faster speed.

Another observation we discovered when implementing the PostgreSQL database system with our website was that ASP .NET does not directly use SQL statements the same way that we would use it in a traditional environments, but rather utilizing LINQ(Language Integrated Query) and ORM (Object-Relational Mapping) libraries such as Entity Framework for PostgreSQL. This abstracts the database layer, and makes us work with the databases in a more object-oriented manner. It might mitigate SQL injection risks since queries are constructed programmatically rather than through string concatenation. However it does have its trade-offs, which we believe or assume to be reduced performance. The paper "The Impact of Object-Relational Mapping Frameworks on Relational Query Performance" [3], describes it to be "the generated SQL might not be as optimized as hand-crafted SQL queries, especially for complex data retrieval operations".

Following our implementation of the initialization of the databases, we also had the observation that each databases initialized their databases with different performances. We observed that Redis was the fastest to populate its database despite the additional query pre-aggregation. We assume this to be due to its in-memory nature and simple data model. Deciding to lay more weight on this assumption we looked at the paper "Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis" [2] which describes the reason for Redis high performance is because of its in-memory storage system, which inherently has much higher throughput and lower latency compared to disk-based storage systems. Following this claim that Redis was observed to be the fastest we observed PostgreSQL to be the slowest among the three databases to populate its tables. This we assume to be because of its reliance on disk-based storage, transactional overhead, and the need to enforce data integrity and schema constraints during data insertion.

4.3 Personal experience with three different Databases

Following the implementation of the three databases we have gotten a lot of different personal experiences and learned a lot new the following three sub chapters will discuses some of the lessens we have learnt when implementing the different databases.

4.3.1 Personal experience with implementing PostgreSQL

Integrating PostgreSQL into our ASP.NET Core MVC application showed us some of the capabilities of relational databases. We also experienced it to be easier to handle and create SQL's, rather than to create the No-SQL databases queries, because of SQL's flexibility in querying, and as well as it being a more familiar logic.

An important insight from this implementation was the recognition of PostgreSQL's need for performance optimization features, such as the use of indexing, partitioning, and query optimization. Understanding how to effectively utilize these features was quite important in enhancing the overall performance of our data queries. Moreover, the initialization phase of our PostgreSQL database was mostly familiar when structuring tables and defining relationships. Which we learnt to be considerably important considering we where using .NET, which is a very strict and structured framework.

4.3.2 Personal experience with implementing MongoDB

Integrating MongoDB into our ASP.NET Core MVC application offered valuable insights into the capabilities of NoSQL databases. MongoDB's flexible document model provided a natural and efficient way to store and manipulate our application's data. The use of MongoDB's aggregation framework for data analysis tasks highlighted the database's power in handling large queries.

One of the key learning's from this implementation was the importance of understanding and leveraging MongoDB's indexing features to optimize query performance. Additionally, the process of initializing the database and managing data insertion underscored the significance of careful planning in data structure and organization.

4.3.3 Personal experience with implementing Redis

Integrating Redis into our application provided valuable lessons on leveraging key-value stores for high-performance data access. Redis's simplicity and speed were important in scenarios requiring rapid data retrieval. The experience highlighted the importance of thoughtful data modeling and aggregation strategies in maximizing Redis's potential. On a more personal note we also learned that performing operations in batches rather than separately, significantly improves the performance.

5 Conclusion

The incorporation of PostgreSQL, MongoDB, and Redis into our web server showed the distinctive advantages and use cases of SQL and NoSQL databases. Populating these databases underscored their unique attributes: PostgreSQL demanded more schema design for structured data, MongoDB facilitated flexible and dynamic data modeling with its document-oriented approach, while Redis excelled in rapid data insertion. Querying mechanisms varied significantly across the databases PostgreSQL was ideal for more complex relational queries, MongoDB supported more data aggregations and larger data, and Redis offered higher speed with key-value data retrieval. This multi implementation of databases not only showed us the specific strengths of each system but also highlighted the necessity of choosing the appropriate database technology based on the application's requirements.

References

[1] Shanshan Chen et al. "Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016, pp. 1660–1667. DOI: 10.1109/TrustCom.2016.0255.

[2] Shanshan Chen et al. "Towards scalable and reliable in-memory storage system: A case study with redis". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE. 2016, pp. 1660–1667.

[3] Derek Colley, Clare Stanier, and Md Asaduzzaman. "The impact of object-relational mapping frameworks on relational query performance". In: *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. IEEE. 2018, pp. 47–52.

[4] Wisal Khan et al. "SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review". In: *Big Data Cogn. Comput.* 7.2 (May 2023), p. 97. ISSN: 2504-2289. DOI: 10.3390/bdcc7020097.

[5] Jaroslav Pokornỳ. "Integration of relational and NoSQL databases". In: *Intelligent Information and Database Systems: 10th Asian Conference, ACIIDS 2018, Dong Hoi City, Vietnam, March 19-21, 2018, Proceedings, Part II 10*. Springer. 2018, pp. 35–45.

[6] *Redis FAQ*. [Online; accessed 21. Apr. 2024]. Apr. 2024. URL: https://redis.io/docs/latest/develop/get-started/faq.

[7] *Replication - MongoDB Manual v7.0*. [Online; accessed 21. Apr. 2024]. Apr. 2024. URL: https://www.mongodb.com/docs/manual/replication.

[8] *Sharding - MongoDB Manual v7.0*. [Online; accessed 21. Apr. 2024]. Apr. 2024. URL: https://www.mongodb.com/docs/manual/sharding.

[9] *What is a Data Federation? | TIBCO*. [Online; accessed 21. Apr. 2024]. Apr. 2024. URL: https://www.tibco.com/glossary/what-is-a-data-federation.

[10] *What is PostgreSQL? – Amazon Web Services*. [Online; accessed 21. Apr. 2024]. Mar. 2024. URL: https://aws.amazon.com/rds/postgresql/what-is-postgresql.

[11] *Why Use MongoDB And When To Use It?* [Online; accessed 21. Apr. 2024]. Apr. 2024. URL: https://www.mongodb.com/why-use-mongodb.

A GitHub

This is the link to the GitHub repository:
`https://github.com/lemoi18/intelligent_data_management`

B Star Schema Data Definition Language

```
using System.ComponentModel.DataAnnotations;
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using Confluent.Kafka.Admin;

namespace Site.Models
{
    public class Sale
    {
        public Sale() { }

        public Sale(string invoiceNo, string stockCode, int quantity,
            double unitPrice, string customerId, string countryId,
            string dateId, string description)
        {
            InvoiceNo = invoiceNo;
            StockCode = stockCode;
            Quantity = quantity;
            UnitPrice = unitPrice;
            CustomerID = customerId;
            CountryID = countryId;
            InvoiceDateID = dateId;
            Description = description;
        }

        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int SalesID { get; set; }
        public string InvoiceNo { get; set; }
        public string StockCode { get; set; }
        public string Description { get; set; }

        public int Quantity { get; set; }
        public double UnitPrice { get; set; }
        public double TotalPrice { get { return Quantity * UnitPrice; } }
        public string InvoiceDateID { get; set; }
        public string CustomerID { get; set; }
        public string CountryID { get; set; }

        // Navigation properties
        public virtual Date Dates { get; set; }
        public virtual Customer Customer { get; set; }
        public virtual Country Country { get; set; }
        public virtual Product Product { get; set; }
```

```
}



public class Date
{
    public Date() { }

    public Date(string dateID , DateTime invoiceDate)
    {
            DateID = dateID;
            InvoiceDate = invoiceDate;
            Year = invoiceDate.Year;
            Month = invoiceDate.Month;
            Day = invoiceDate.Day;
    }


    [Key]
    public string DateID { get; set; }
    public DateTime InvoiceDate { get; set; }
    public int Year { get; set; }
    public int Month { get; set; }
    public int Day { get; set; }
}

public class Product
{
    public Product() { }

    public Product(string stockCode, string description,
        double unitPrice)
    {
        StockCode = stockCode;
        Description = description;
        UnitPrice = unitPrice;
    }

    [Key] // Define a primary key for the Product entity
    public string StockCode { get; set; }
    public string Description { get; set; }

    public double UnitPrice { get; set; }

}

public class Customer
```

```
    {
        public Customer() { }

        public Customer(string customerId)
        {
            CustomerID = customerId;
        }

        public string CustomerID { get; set; }
    }

    public class Country
    {
        public Country() { }

        public Country(string countryId, string countryName)
        {
            CountryID = countryId;
            CountryName = countryName;
        }

        public string CountryID { get; set; }
        public string CountryName { get; set; }
    }
}



public class SalesByProductViewModel
{
    public string StockCode { get; set; }
    public int Quantity { get; set; }
    public double UnitPrice { get; set; }
    public double TotalPrice { get; set; }

}

public class TotalSalesByCountryViewModel
{
    public string Country { get; set; }
    public double TotalSales { get; set; }
}

public class CustomerLifetimeValueViewModel
{
    public string CustomerID { get; set; }
    public double LifetimeValue { get; set; }
```

```
}
public class InvoiceSummaryViewModel
{
    public string InvoiceNo { get; set; }
    public double TotalAmount { get; set; }
    public string CustomerName { get; set; }
    public string CountryName { get; set; }
    public DateTime InvoiceDate { get; set; }
    // ... add other necessary fields
}
public class SalesTrendViewModel
{
    public int Year { get; set; }
    public int Month { get; set; }
    public double TotalSales { get; set; }
}
```