Group 2:
Frederik Hallengreen Hansen
Albert Bek Jørgensen
Jakob Nikolaj Alstrup Helander
Daniel Starck Lorenzen

Sparehub - Databases for Developers

Delivery Date: 18/12-2024

Github link: https://github.com/FHallengreen/sparehub

List of figures 4
1. Introduction 7
1.1. Problem description (+ schema of the whole system) 8
1.2. Explanation of choices for databases and programming languages, and other tools. 10
1.3 Description of system architecture 11
1.4 Brief description of CRUD, Service layer for our application 11
2. Relational database 12
2.1. Intro to relational databases 12
2.2. Database design 12
2.2.1. Entity/Relationship Model (Conceptual -> Logical -> Physical model) 13
Conceptual Model (figure 1) 13
Logical Model (figure 2) 13
Physical Model (figure 3) 14
2.2.3. Normalization process 15
Objectives of Normalization 15
Normalization in Our Database Design 15
2.3.1. Data types 16
Numeric Types 16
String Types 16
Boolean 17
Other Types 17
2.3.2. Primary and foreign keys 18
2.3.3. Indexes 18
2.3.4. Constraints and referential integrity 18
2.4. Stored objects – stored procedures / functions, views, triggers, events 19
2.5. Transactions. Explanation of the structure and implementation of transactions 20
2.6. Auditing. Explanation of the audit structure implemented with triggers 21
2.7. Security. 21
2.7.1. Explanation of users and privileges 22
2.7.2. SQL Injection – what is it and how is it dealt with in the project? 23
2.8. Description of the CRUD application for RDBMS – security – registration / login, transactions, etc. 24
3. Document database 26
3.1. Intro to document databases 26
3.2. Database design + features like indexes, PKs, constraints, etc. 27
3.3. Description of the CRUD application for the document database – security –

registration / login, transactions, etc. 29
4. Graph database 32
4.1. Intro to graph databases 32
4.2. Database design + features 33
4.3. Description of the CRUD application for the document database – security – registration / login, transactions, etc. 34
5. Conclusions (discussing the similarities and differences between used database types) 40

List of figures

Figure 1 - Conceptual Model.

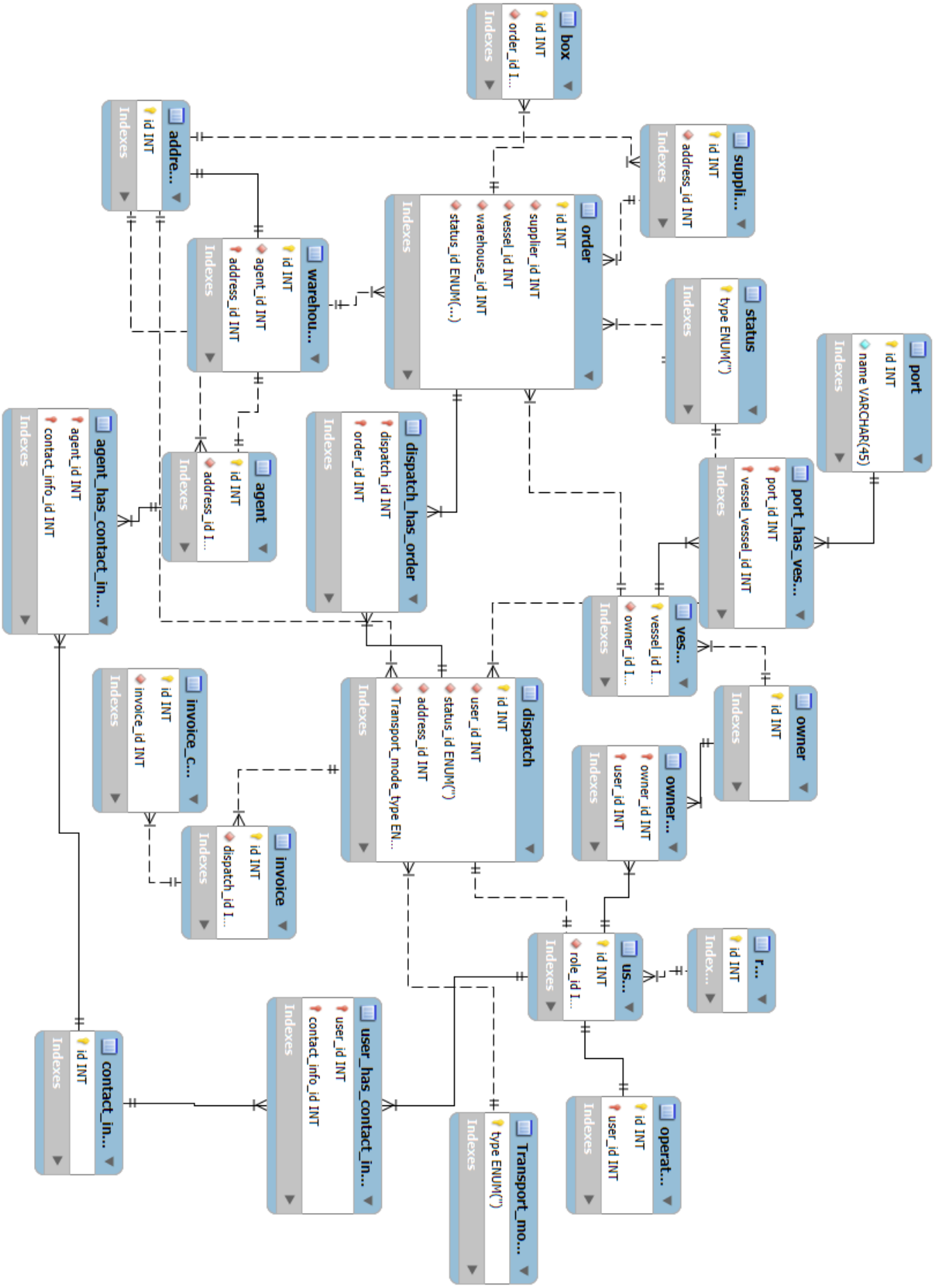
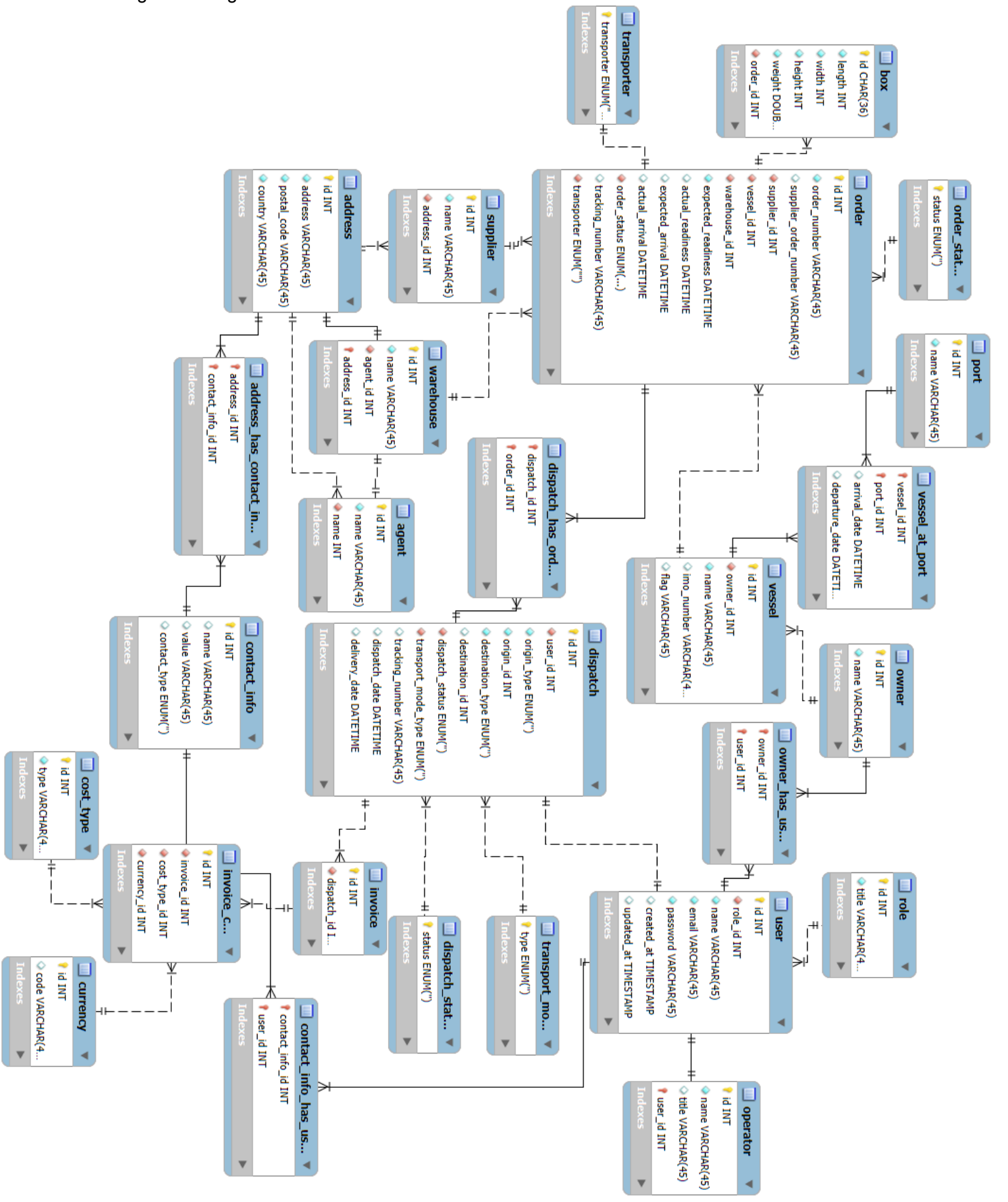


Figure 2 - Logical Model



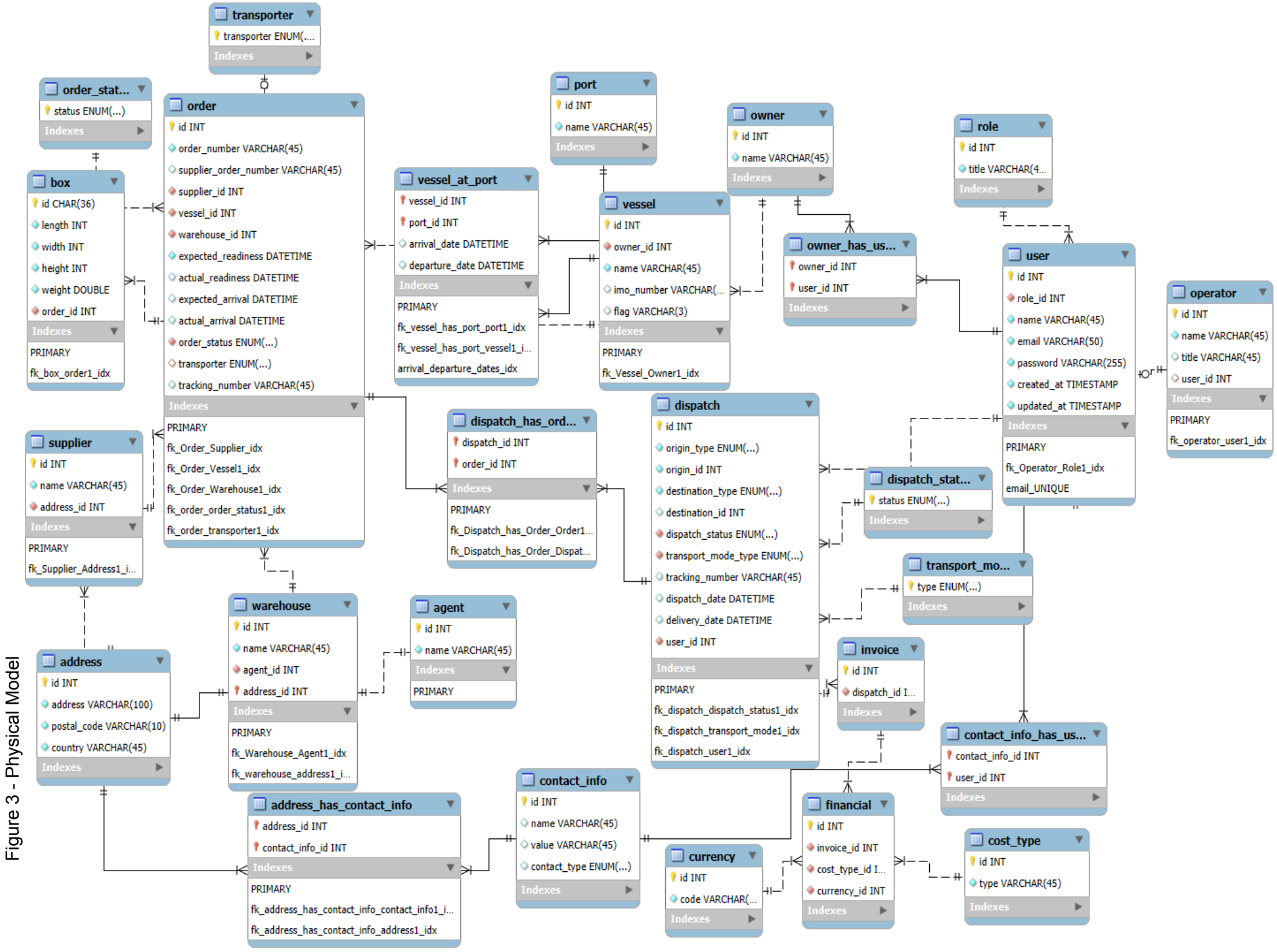


Figure 3 - Physical Model

1. Introduction

In the global maritime industry, the maintenance and operation of commercial vessels are heavily dependent on the timely and efficient handling of spare parts. The complexity of managing these parts sourcing from multiple suppliers, tracking deliveries to various warehouses, and ensuring timely arrival on vessels, presents significant logistical challenges. Strict timelines, high operational stakes, and large volumes of data emphasize the need for robust and reliable data management solutions.

One of our group members has firsthand experience working daily with the transport of ship spares. From this professional perspective, it is evident that the current processes, often supported by legacy systems or cumbersome spreadsheets, could greatly benefit from a more modern, streamlined database solution. Such improvements would enhance efficiency, data quality, and user experience, while reducing manual data entry and simplifying reporting and decision-making.

Our project, therefore, initially focuses on designing and implementing a relational database system (RDBMS) called SpareHub, dedicated to managing the flow of spare parts. This forms our foundation.

However, as part of the project requirements and to simulate a realistic scenario where a company might consider changing or supplementing its database technology, we will implement the same or similar functionalities using three different database technologies:

1. RDBMS (MySQL): Establish a structured, normalized schema for reliable data integrity.
2. Document Database (MongoDB): Handle flexible, nested data (e.g., box details) to demonstrate how schema-less structures can streamline certain data models.
3. Graph Database (Neo4j): Represent and query complex relationships (e.g., vessel-port connections) more naturally than traditional relational models, revealing how graph queries can simplify certain operations.

By implementing the same system across these three technologies, we can compare their strengths and weaknesses, understanding where each database type excels and where it falls short. This approach mirrors real-world situations in which organizations may start with one technology and later explore others to meet evolving needs. The comparison will highlight key trade-offs, performance

differences, scalability considerations, and how certain parts of the data model are more intuitively expressed in different database paradigms.

Even within the limited time available, this multi-database strategy not only meets the educational objective of experimenting with various technologies but also offers valuable insights into how flexible, future-facing database solutions can be integrated to improve system reliability, efficiency, and scalability in complex maritime logistics operations.

1.1. Problem description (+ schema of the whole system)

Problem context:
A typical workflow for an operator in Sparehub involves monitoring incoming orders and sometimes arranging collections from suppliers all around the world. Then dispatches are arranged to the vessel wherever it is in the world arriving timely to have the orders delivered onboard. It is very crucial that the database is trustworthy and well structured to avoid any issues related to the vessel orders. Any errors could result in huge extra costs for the vessels if parts are not received in time.

Key Entities and Relationships:

Address & Associated Entities:

Address records form a core reference point, storing physical location details. Multiple entities reference address_id to avoid redundancy:

- Supplier: Each supplier is linked to an address, ensuring that supplier locations remain consistent and easily updatable.
- Warehouse: Each warehouse references an address as well, allowing multiple facilities (warehouses) to be uniquely tied to distinct locations.

Owner & Vessel:
Vessels are central to the maritime spare parts supply chain. Each Vessel record belongs to a specific Owner. This relationship ensures that all vessel-related data—such as its IMO number and flag—is correctly associated with the right owning entity. Vessels may also have scheduled visits to various ports, tracked separately.

Agent & Warehouse:
Agent entities represent logistics handlers at specific locations. Each Warehouse is managed by an agent and is associated with a physical address. Warehouses hold spare parts and serve as collection or distribution points within the supply chain.

Order & Related Details (Supplier, Vessel, Warehouse, Status, Transporter):
Order entities link suppliers, vessels, and warehouses:

- Supplier and Vessel: An order specifies which supplier the parts are coming from and which vessel they are intended for.
- Warehouse: Orders may route parts through a designated warehouse, either as a temporary storage location or a point of consolidation.

Order Status and Transporter: Each order references order_status (e.g., Pending, Delivered) and a transporter (like DHL, FEDEX, GLS) along with a tracking number.

Dispatch, Dispatch Status, and Transport Mode:
Dispatch records capture the movement of orders (or subsets of orders) from one location to another be it a warehouse, supplier, or address:

- Dispatch Status: Tracks the current state of a dispatch (Created, Sent, Delivered).
- Transport Mode: Identifies whether the dispatch is carried out by Air, Sea, or Courier.
- User Linkage: Each dispatch references a user_id indicating which user oversaw or executed the dispatch process.
  A many-to-many relationship between dispatches and orders is maintained via the dispatch_has_order join table, linking specific dispatch movements to the underlying orders they carry.

Invoice & Financial Details (Cost Type, Currency):
Invoice records are tied to dispatches, representing the billing side of a shipment. Financial details such as shipping, handling, customs, and insurance costs are stored in the Financial table:

- Cost Type and Currency: Each financial record references a cost_type (e.g., Shipping, Handling) and a currency (e.g., USD, EUR), enabling comprehensive financial overviews and easy global transactions.

Box & Order Integration:
Box entities represent individual packages associated with an order. They store

dimensions, weight, and are linked via order_id to ensure traceability of each physical item shipped as part of an order.

Ports & Vessel at Port:
Port entities store information about geographic ports. The vessel_at_port table links vessels to ports, along with arrival and departure times, supporting historical and predictive analytics regarding vessel movements.

Roles, Users, and Security (Operator):

- Role & User: Users in the system have assigned roles (e.g., Administrator, Operator, Viewer), controlling their permissions and access rights.
- Operator: Operators are a specialized type of user with additional attributes, indicating their responsibility levels or titles.
- Security Model: The database enforces secure access through roles and users, ensuring that certain actions like inserting, updating, or reading sensitive data are only performed by authorized individuals or services.

Contact Information & Linking Entities to Users:
Contact_info and related junction tables (e.g., address_has_contact_info and contact_info_has_user) allow multiple contacts (email, phone, mobile) to be associated with addresses and users. This supports robust communication channels and streamlined user interactions.

The diagram would show all the main tables (e.g., order, supplier, vessel, warehouse, address, agent, invoice, financial, dispatch, box, contact_info, role, user), their primary keys, foreign keys, and relationships. This provides a visual summary of how the system's data is interconnected.

1.2. Explanation of choices for databases and programming languages, and other tools.

We chose MySQL because it's the database we've worked with the most during our Datamatiker education, and it's the one we used in classes, so we feel comfortable and confident with it. MySQL is also one of the world's most popular and user-friendly databases, making it a perfect fit for our needs. Paired with MySQL Workbench, we have a straightforward way to design and manage our schema.
On the backend, we're using C# and Entity Framework. EF simplifies interacting with MySQL by letting us work with our data through code instead of writing a lot of

SQL queries by hand which is also not considered safe. This approach makes it easier to quickly adjust our data model and adapt as our project grows.
For our document database, we chose MongoDB since it's the one we're most familiar with from our classes. Similarly, for the graph database, we decided to use Neo4J because it's what we practiced with and understand best.

1.3 Description of system architecture

Our application consists of a REST API (controller) with associated service layers. These two layers are designed to be generic and abstracted from the concept of our three databases. The dependency on the databases is managed within the repository layer, which will be described in detail for each type of database.

This architecture allows us to develop a single application that supports all three databases, eliminating the need to create separate applications for each database. This flexibility is achieved by using interfaces and a factory-like class that dynamically provides the appropriate repository based on the database type.

For each repository we create (e.g., WarehouseRepository), we implement three distinct repositories, each capable of communicating with its respective database. The differences between the databases are too significant to share a single repository, making separate implementations necessary for each.

1.4 Brief description of CRUD, Service layer for our application

Our application implements a standard layered architecture to ensure modularity, scalability, and maintainability. The REST API serves as the primary interface between clients and the application, exposing endpoints that adhere to RESTful principles. Endpoints are resource-oriented and leverage standard HTTP methods (GET, POST, PUT, DELETE) for operations. The API layer is responsible for request validation, authentication, authorization, and formatting responses with appropriate HTTP status codes. It ensures client-facing responsibilities remain isolated from the underlying business logic.

The service Layer acts as an intermediary between the API and the repository layer, encapsulating business logic and responsible for converting objects to the proper format. This layer enforces application constraints, handles exceptions, and coordinates operations across multiple repositories when needed.

2. Relational database

2.1. Intro to relational databases

In a complex environment like spare parts logistics, where the timely availability of spare parts can make or break a vessel's operational schedule and organization is paramount. Relational databases offer a structured way to store and manage data, using tables to represent entities such as orders, suppliers, warehouses, and vessels. Each table consists of rows (records) and columns (attributes) that define and maintain consistent formats for the information they hold.

At the core of the relational model is the concept of relationships - links between tables that reflect real-world associations. For instance, an order is linked to a supplier who provides the spare parts and to a warehouse where those parts may be temporarily stored. By using primary keys to uniquely identify records and foreign keys to reference related records in other tables, relational databases ensure that the data remains coherent and meaningful. This referential integrity is crucial for preventing contradictory or orphaned entries, a risk that would be costly and confusing in a high-stakes scenario like for our application.

In the context of our SpareHub application, the relational approach allows us to define clear boundaries between different types of information. This makes it easier to track which supplier provided which parts, which warehouse currently stores them, and which vessel is scheduled to receive them. As a result, users can trust the data's accuracy. This reliability, combined with a strict schema and well-defined constraints, ensures that the relational database is not just a convenient tool, but a vital backbone for handling the complexity, volume, and urgency of spare parts logistics data.

2.2. Database design

The database design process began with a thorough review of the application's requirements. Since one of our team members works with transport of spare parts on a daily basis, we had a clear understanding of the real-world entities, relationships, and workflows involved. Their insight made it straightforward to identify which tables, attributes, and constraints were needed to accurately model the data and support the intended functionality of the system.

2.2.1. Entity/Relationship Model (Conceptual -> Logical -> Physical model)

Conceptual Model (figure 1)

Before diving into the technical aspects of data types & keys, we began by forming a conceptual model of our logistics application. This initial stage focused on identifying the essential entities and their natural relationships - much like what established solutions already provide. Our goal was to map out the domain in a way that reflects the real-world complexity of managing and monitoring ship spares orders for vessel operators and owners.

In this conceptual phase, we avoided technical specifics like data types. Instead, we concentrated on the main entities and how they interact. For example, we identified core entities like Supplier, Vessel, Warehouse, Order, Dispatch, Agent, and Port, each representing a different aspect of the supply chain. Suppliers deliver the ship spares(order), vessels require these items for ongoing operations, warehouses serve as storage or transit points, agents manage warehouse activities, dispatches consolidate multiple orders into one shipment for the vessel and ports represent geographic points in a vessel's journey where these shipments are sent to.

By outlining these entities and their interconnections such as an Order linking a Supplier to a specific Vessel, or a Warehouse managed by an Agent we established a clear understanding of the domain. This conceptual model helps ensure we're capturing the fundamental logic and workflows that operators and vessel owners rely on. It also provides a strong foundation for subsequent modeling phases, allowing us to confidently proceed with logical and physical design steps that will bring this logistics application closer to a robust, real-world-ready system.

Logical Model (figure 2)

The logical model is the detailed blueprint of our database design, serving as the intermediary step between the conceptual model and the physical implementation. It captures all the necessary details to define the structure and relationships of the database while remaining independent of the specific database management system.

Tables and Their Attributes

Each table in the logical model represents a real-world entity or relationship relevant to the logistics and operations of ship spare parts. Key details include:

- **Order Table**: Stores details about orders, such as order numbers, supplier references, vessel references, warehouse destinations, and status (e.g.,

Pending, Delivered). Attributes such as expected and actual dates allow tracking of order progress.

- **Supplier Table**: Maintains a record of suppliers, including their unique IDs and associated addresses.
- **Vessel Table**: Captures vessel information, including its IMO number, flag, and associated owner. This ensures traceability for all orders linked to specific vessels.
- **Warehouse Table**: Tracks warehouse information, linking each warehouse to an agent and its physical address.
- **Dispatch Table**: Manages the movement of orders, capturing details such as origin, destination, transport mode, and dispatch status.

Normalization in the Logical Model

The logical model also involves normalization, to eliminate redundancy and ensure data consistency:

- All attributes are atomic, fulfilling the requirements of 1NF.
- Each table contains attributes that are functionally dependent on its primary key, adhering to 2NF.
- Non-key attributes depend only on the primary key, ensuring compliance with 3NF.

Physical Model (figure 3)

The physical data model translates the logical design into a structure that can be implemented in MySQL. At this stage, we finalized the technical details required for deployment, including data types, primary and foreign keys, indexes, and constraints.

The physical model ensures that the system is efficient and reliable for handling real-world data loads and queries. Each table was carefully designed with the following considerations:

- **Data Types:** Choosing the most suitable types (e.g., INT, VARCHAR, DATETIME) for each attribute to optimize storage and performance.
- **Keys and Indexes:** Defining primary and foreign keys for relationships, and creating indexes on frequently queried columns to improve read performance.

- **Constraints:** Enforcing business rules and referential integrity using constraints like NOT NULL, UNIQUE, and FOREIGN KEY.

The physical model represents the final blueprint for implementing the database, bridging the gap between design and deployment. Details of the above considerations are explained in more detail in upcoming section 2.3.

2.2.3. Normalization process

The normalization process focuses on organizing the database structure to eliminate redundancy, ensure data consistency, and optimize storage efficiency. This step was essential in designing a reliable and scalable database for our logistics application.

Objectives of Normalization

- **Reduce Data Redundancy:** Ensure that data is not duplicated across tables, minimizing storage requirements and the risk of inconsistencies.
- **Improve Data Integrity:** Enforce clear relationships between tables, so changes in one table are correctly reflected in others.
- **Simplify Maintenance:** Make the database easier to update, expand, and adapt to future requirements.

Normalization in Our Database Design

The database schema was normalized following standard normalization rules to ensure a clear, consistent, and efficient design.

1. **First Normal Form (1NF)**
    - **Atomic Values:** All fields contain indivisible values. For example, addresses were split into separate columns for address, postal_code, and country.
    - **Unique Rows:** Each table has a primary key to uniquely identify each record, such as id in the supplier and vessel tables.
    - **No Repeating Groups:** Data such as multiple contact methods (email, phone) were stored in a separate contact_info table instead of as multiple columns in the same table.
2. **Second Normal Form (2NF)**
    - **Removal of Partial Dependencies:** Non-key attributes are fully dependent on the entire primary key. For example:
        - The warehouse table stores only information related to warehouses, such as their name and address_id.

        - Agent details are stored in the agent table, avoiding duplication in the warehouse table.
    - Many-to-many relationships, such as between dispatch and order, were resolved using a join table (dispatch_has_order).
3. **Third Normal Form (3NF)**
    - **Elimination of Transitive Dependencies:** Non-key attributes depend only on the primary key. For instance:
        - The vessel table includes details like imo_number and flag but does not store information about the vessel owner, which is stored separately in the owner table.
        - Financial details like cost_type and currency are stored in their respective tables and referenced via foreign keys in the financial table.

2.3.1. Data types

In relational databases like MySQL, there are multiple data types that support common use cases for developers. The most basic types are numbers, strings, and booleans. MySQL also supports other data types such as date/time and JSON.

Numeric Types

Numeric values include integers, decimals, and floats. Subcategories of each type, such as TINYINT, SMALLINT, and BIGINT for integers, differ primarily in the amount of data they can store (measured in bytes) and their range. For instance, BIGINT is a good choice for a primary key in tables with potentially large numbers of rows, as it can store very large integers.

Decimals and floating-point numbers are slightly similar but have key differences:

- **Decimals**: Have fixed precision and are often used for financial calculations where exactness is crucial. The most common type is DECIMAL.
- **Floating-Point Numbers**: Used for a wider range of values but with less precision. They are typically used for scientific or engineering calculations where very small or very large numbers are common.

String Types

String types vary based on the amount of storage required and the specific use case.

- **CHAR**: Used for fixed-length strings. The length must be specified, and all stored values will occupy that length, even if they are shorter. An example use case could be storing currency or country codes.
- **VARCHAR**: The most commonly used string type in SQL. The maximum length is specified, but stored values can vary in length up to the defined maximum. This makes it more space-efficient for variable-length data like names or email addresses.
- **TEXT** and **BLOB**: These are designed for larger data.
    - **TEXT**: For very large strings, such as paragraphs or logs.
    - **BLOB**: For binary data, such as images, audio files, or other non-text files.

Boolean

Booleans represent TRUE or FALSE. In MySQL, booleans are stored as TINYINT (1 byte), where 1 represents TRUE and 0 represents FALSE.

Other Types

MySQL also supports several specialized data types:

- **Date/Time Types**: Include DATE, TIME, DATETIME, TIMESTAMP, and YEAR. These are essential for applications involving scheduling, logging, or tracking temporal data.
- **Geolocation Types**: Support spatial data such as POINT, LINESTRING, and POLYGON, which are used in geospatial applications.
- **JSON**: Enables storage and manipulation of structured JSON data, which is useful for semi-structured or hierarchical data.
- **ENUM**: Represents a predefined list of allowed string values, which can be useful for fields like status or category.

What we use

We primarily use **string** and **numeric types** in our tables but also utilize more specific types like **ENUM** for attributes such as destination_type and transport_mode_type. In these cases, it makes sense to restrict the values to predefined options.

We use the **DATETIME** type when storing attributes like delivery_date to represent specific dates and times accurately.

For specific string data, we use **CHAR(36)** for the box_id. Since the box ID is always exactly 36 characters long, it is efficient and appropriate to use a fixed-length type like CHAR.

We frequently use the **VARCHAR(n)** type because we have implemented validation using **Entity Framework (ORM)**. Here, we often specify the exact lengths of table attributes, which results in consistent usage of the VARCHAR(n) type. Examples of this could be username and password.

2.3.2. Primary and foreign keys

Primary keys are the unique identifier for a specific entity in the table. This is required to access the entity to make e.g: crud operations. Foreign keys are used to map relationships between entities and are the primary key from a foreign table. Thus a “foreign” key.

2.3.3. Indexes

Indexes are a powerful tool that can be used in relational databases. They can be created on columns to make it easier to filter data. Adding indexes is a great way to increase read performance, although it can decrease write performance. Indexes are also always present on the primary key within tables. One reason for this is that the primary key must be unique. Indexing the primary key ensures that the database does not allow duplicate IDs, assuming the primary key is the ID.

An example of this is the `vessel_at_port` table. In this table, we have a column called `arrival_date`, which stores the date a vessel arrives at a specific port and is frequently used for search operations. By creating an index on `arrival_date`, we can efficiently filter rows for queries where we search for vessels within a particular date range. For example, if we want to display all vessels that arrived between March 1st, 2024, and April 30th, 2024, the index ensures that the database can quickly and efficiently locate the relevant rows without scanning the entire table.

2.3.4. Constraints and referential integrity

Constraints are rules applied to columns in a relational database to ensure the accuracy, consistency, and integrity of the data. They create validation conditions that must be met before data is saved or updated in the database. Several types of constraints are commonly used in relational databases:

Domain Constraints
Domain constraints restrict the values in a column by ensuring they match the specified data type. For example, if a column is assigned the int data type, this constraint ensures that only integers can be added to the column.

Entity Integrity Constraints
Entity integrity constraints ensure that the primary key cannot be null. This is essential because the primary key uniquely identifies each record in the table.

Referential Integrity Constraints
Referential integrity constraints maintain consistency among relations between tables. These constraints are typically enforced through foreign keys, ensuring that relationships between tables remain valid by associating entries in one table with corresponding entries in another.

Key Constraints
Key constraints ensure that designated key columns contain unique values across all rows in a table. This is crucial for maintaining data uniqueness and integrity.

Check Constraints
Check constraints validate column values against specific conditions. Unlike domain constraints, which check data types, check constraints can enforce conditions such as ensuring a value is greater than zero or another specified number.

Not Null Constraints
Not null constraints ensure that a column cannot contain null values. This is useful for columns where data must always be present in every row.

2.4. Stored objects – stored procedures / functions, views, triggers, events

Stored procedures/functions are SQL statements that are stored in the database and precompiled for reuse. They encapsulate logic that can be executed by calling either a procedure or a function. While similar, there are key differences between stored procedures and stored functions.

A **stored procedure** is a sequence of SQL statements that can perform various operations such as querying, inserting, updating, or deleting data. Stored procedures can take input parameters and return either single values or result sets. They are primarily used to automate repetitive database operations, improve

performance, and centralize business logic.
For example, we could use a stored procedure to query `non_active_orders`, which contains orders that are no longer active. A stored procedure could filter these orders based on conditions such as an order status of `'Cancelled'` or `'Delivered'` and return only the relevant data. This avoids duplicating the same SQL logic across the application.

A **stored function** is similar to a stored procedure but always returns a single value. For instance, we could create a function to calculate the total number of boxes in an order. By passing the `order_id` as input, the function would compute and return the total number of boxes for that order.

A **view** is a virtual table that simplifies complex queries by abstracting them into a reusable query structure. For example, we can create a view for `non_active_orders` to encapsulate the logic for retrieving inactive orders. This allows us to query the view directly without repeatedly writing the same conditions.

Triggers automate actions in response to specific table events, such as `INSERT`, `UPDATE`, or `DELETE`. For example, a trigger could log changes to the `dispatch` table whenever updates occur. This ensures that important changes are tracked automatically.

Events are tasks scheduled to run at specific intervals, automating routine database operations such as cleanup or archiving. For instance, we could create an event that runs daily and deletes `non_active_orders` with a status of `'Cancelled'` that are older than 30 days. This keeps the database clean and optimized.

2.5. Transactions. Explanation of the structure and implementation of transactions

A transaction is a sequence of SQL operations performed as a single unit. Transactions play a crucial role in maintaining data integrity and ensuring that operations are executed correctly. They guarantee that all operations either succeed together or have no effect in case of failure.

In the Sparehub database, transactions can be implemented to ensure consistency when performing multiple related operations. For example, consider order creation and its association with a dispatch. Transactions can ensure that either both operations succeed or neither is executed. Let’s say we are creating a new order and a corresponding dispatch—transactions ensure that the dispatch is inserted only if the order is successfully created.

2.6. Auditing. Explanation of the audit structure implemented with triggers

Auditing in a relational database is the process of tracking changes to data within the database. In our database, auditing can be implemented using triggers, which automatically log changes to tables during events such as `INSERT`, `UPDATE`, or `DELETE`.

In an audit structure, changes to key tables—such as `order`, `dispatch`, or `vessel`—are logged into dedicated audit tables. These audit tables typically record what changed, when it changed, and who made the change.

For example, consider the `dispatch` table. For any `UPDATE` operation on `dispatch`, we could log the changes into a `dispatch_audit` table. To achieve this, a trigger can be created to execute after any `UPDATE` on the `dispatch` table, automatically logging the changes into the `dispatch_audit` table. This approach can then be extended to other operations and other tables as needed.

2.7. Security.

Security in a relational database is critical to protecting sensitive data and ensuring privacy, as well as preventing unauthorized access or breaches. There are several standard practices designed to safeguard a database at multiple levels.

Access Control and Role-Based Access Control (RBAC)
Access control, including role-based access control (RBAC), secures the database at the user access level. It is essential to manage who can access the database and define the permissions they have. For instance, a data analyst may only need "read-only" access, whereas an admin would require full access. By implementing RBAC, organizations can ensure users have access only to the data and functions necessary for their role.

Authentication
Authentication enhances database security by requiring users to prove their identity. This can be achieved through strong password policies and multi-factor authentication (MFA). Enforcing these measures helps prevent unauthorized individuals from breaching the database using weak or compromised credentials.

Data Encryption
Data encryption is a standard security measure that protects sensitive information such as passwords, files, credit card numbers, and other private data. By

encrypting data, even if someone gains unauthorized access, they cannot view the actual values. Failing to encrypt sensitive data is a significant security risk.

Data Masking
Data masking obscures sensitive data for non-privileged users. For example, a value like “123-345-678” might be displayed as “XXX-XXX-678”. This is known as dynamic data masking, where sensitive data is hidden in real-time based on user access. Static data masking, on the other hand, permanently anonymizes data in non-production environments, ensuring sensitive information is not exposed during testing or development.

2.7.1. Explanation of users and privileges

In relational databases, users and their associated privileges play a vital role in securing the database and controlling access to both the data and operations. Properly managing users and privileges ensures that only authorized individuals can interact with the database.

Privileges define which actions a user is allowed to perform. These actions include operations such as `SELECT`, `INSERT`, and `UPDATE`. Privileges can also be assigned at various levels:

- **Global level**: Permissions apply to all databases.
- **Database level**: Permissions apply to a specific database.
- **Table level**: Permissions apply to specific tables.
- **Column level**: Permissions apply to specific columns within a table.

Some general best practices for user management include:

- Granting only the minimum privileges required for each role.
- Avoiding the use of `ALL PRIVILEGES` unless absolutely necessary.
- Enforcing strong passwords for all users.

By managing users and privileges effectively, we can control who has access to specific data and what actions they are allowed to perform. This approach improves security by ensuring that privileges are granted only as necessary for each user or role.

2.7.2. SQL Injection – what is it and how is it dealt with in the project?

SQL injection is a type of security vulnerability that allows attackers to manipulate a database query by injecting malicious SQL code into user inputs. This can lead to unauthorized data access, data modification, or even deletion of critical records. In applications where sensitive data is stored, such as user credentials or order details, preventing SQL injection is a critical part of database security.

How SQL Injection is Prevented in This Project:

Sanitization in the Frontend:

To minimize risks, user inputs are sanitized on the frontend before being sent to the backend. This involves validating inputs to ensure they conform to expected formats (e.g., email addresses, numeric values) and rejecting any unexpected or suspicious data. While frontend sanitization is helpful, it is not the sole defense, as attackers can bypass the frontend entirely.

Entity Framework in the Backend:

In the backend, all database interactions are handled through Entity Framework (EF). EF is an Object-Relational Mapping (ORM) tool that abstracts raw SQL queries and prevents SQL injection by using parameterized queries. Instead of constructing raw SQL strings with user inputs, EF ensures that all inputs are treated as parameters, making it impossible for injected SQL code to be executed.

This approach significantly reduces the likelihood of SQL injection attacks, as EF automatically validates inputs before sending them to the database.

Secure Login with Hashed Passwords:

For user authentication, we use BCrypt to hash passwords before storing them in the database. This ensures that even if the database is compromised, plaintext passwords are not exposed. During login, the entered password is hashed and compared with the stored hash, providing an additional layer of security.

By not using raw SQL queries for authentication and relying on EF and hashed passwords, we eliminate a common vector for SQL injection attacks.

2.8. Description of the CRUD application for RDBMS – security – registration / login, transactions, etc.

Entity generation

As mentioned, we use **Entity Framework** for C#. With this, we can use **ORM (Object-Relational Mapping)** to create our tables. We simply define model classes in C# to generate the tables. In a simple entity like **Address**, which contains attributes such as **AddressLine**, **PostalCode**, and **Country**, we need to specify the primary key by doing the following:

```
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public int Id { get; init; }
```

Now the database recognizes this as the primary key and will use it as its reference. The other attributes are automatically added to the entity as columns in the table. Entities that reference other entities (foreign keys) follow similar logic to generate the table.

For example, a **Warehouse** entity contains a reference to an **Agent**.

```
[ForeignKey("AgentId")]
public required AgentEntity  Agent { get; init; }
```

Querying

To query data in SQL, we use the DbContext class. This class is used to execute SQL commands, abstracting the need to write raw SQL code. The advantage of this approach is that it is generally simpler and faster to write SQL queries. However, it does have some disadvantages, such as reduced customizability, which could potentially impact performance. More importantly, the abstraction can sometimes make the code less understandable. An abstraction can create confusion about what is actually happening behind the scenes.

That said, we believe it is more effective to use this framework rather than writing raw SQL, especially since our application is relatively simple and mostly consists of basic CRUD operations.

Below, we see a simple function to get all addresses in the database. Here, we use the built-in function to fetch all entities in the table.

```
public async Task<List<Address>> GetAddressesAsync()
{
   var addresses = await dbContext.Addresses.ToListAsync();
   return mapper.Map<List<Address>>(addresses);
}
```

Below, we see a function to create a new entity in the **Address** table. It's still very simple and fast to implement. We use mapping to format the objects properly before or after modifying the data.

```
public async Task<Address> CreateAddressAsync(Address address)
{
   var addressEntity = mapper.Map<AddressEntity>(address);
   await dbContext.Addresses.AddAsync(addressEntity);
   await dbContext.SaveChangesAsync();
   address.Id = addressEntity.Id.ToString();

   return address;
}
```

Below, we see a slightly more complicated query, but it is still relatively straightforward. In this case, we are fetching orders, which reference other entities. The query should include all the related entities as well. In general, if we are only working with basic CRUD operations, it doesn't get more complicated than this. This is one of the major advantages of using this type of framework.

```
public async Task<Order?> GetOrderByIdAsync(string orderId)
{
   var orderEntity = await dbContext.Orders
        .Include(o => o.Supplier)
        .Include(o => o.Vessel)
        .ThenInclude(v => v.Owner)
        .Include(o => o.Warehouse)
        .ThenInclude(w => w.Agent)
        .Include(o => o.Boxes)
        .AsNoTracking()
        .FirstOrDefaultAsync(o => o.Id.ToString() == orderId);
```

```
    return orderEntity != null ? mapper.Map<Order>(orderEntity) :
null;
}
```

Security

Using the EF framework with LINQ, we automatically use parameterized methods to query data, which helps protect against SQL injection. We also implement user input sanitation to validate the format of the attributes in our entities. This ensures that the server always receives data in the correct format. For example, we can enforce rules such as the length of the username or password:

```
[Required]
[MaxLength(45)]
public required string Password { get; set; }
```

We use DTOs (Data Transfer Objects) to ensure that we return data in the desired format. With DTOs, we can specify that the server should only return the necessary data to the client, rather than everything. This approach is particularly useful when there is sensitive data that we don't want the server to expose to the client. A DTO is simply a class with attributes that define the structure of the data we want to return.

3. Document database

3.1. Intro to document databases

Document databases are a type of NoSQL database designed to store, manage, and retrieve data in a document-oriented format, typically using JSON, BSON, or XML. Unlike traditional relational databases, document databases use a flexible schema, allowing for dynamic and varied data structures within a single collection. This makes them ideal for applications requiring high scalability, quick iterations, and the ability to handle unstructured or semi-structured data. As mentioned, we use MongoDB.

3.2. Database design + features like indexes, PKs, constraints, etc.

Collection Structure

The database design for MongoDB consists of separate collections for entities that are largely static and reusable across multiple orders. These collections include:

- Orders
- Suppliers
- Agents
- Warehouses
- Vessels
- Owners
- Boxes
- Dispatches

Each collection contains documents that represent individual entities. The Orders collection references other collections using foreign keys (e.g., supplierId, warehouseId). Below example how orderCollection is implemented using references to the other tables:

```
{
  "_id": "order_1234",
  "orderNumber": "ORD-5678",
  "supplierNumber": SUP-223,
  "supplierId": "1",
  "warehouseId": "2",
  "vesselId": "3",
  "expectedReadiness": "2024-06-15T10:00:00Z",
  "actualReadiness": "2024-06-16T12:00:00Z",
  "transporter": "DHL",
  "status": "stock",
  "trackingNumber": "TRACK-98765"
}
```

Primary Keys (PKs):

Each collection in the database uses MongoDB's default **_id** field as the primary key. The _id field is automatically generated as an **ObjectId**, ensuring global

uniqueness for each document across collections. This guarantees efficient identification and retrieval of documents.

ObjectId is a 12-byte identifier that includes:

- A timestamp for creation.
- A machine identifier.
- A process ID.
- An incrementing counter to ensure uniqueness.

Example:

- Orders: _id (ObjectId("60adf2c5e7b3a6b4d3e58f9c")
- Box: _id ObjectId("60adf2c5e7b3a6b4d3e58f9e")

Benefits of ObjectId:

- Uniqueness: Ensures no conflicts across collections.
- Performance: Optimized indexing on _id by default.
- Automatic Generation: No need for manual management of keys.

Indexes:

Indexes are critical for query performance. MongoDB automatically creates an index on the _id field for every collection. Additional indexes are added based on query requirements to optimize performance.

Key Indexes:

1. **Default _id Index**:
    - Automatically created for efficient lookups on _id.
2. **Foreign Key Indexes**:
   To optimize queries involving referenced documents, it makes sense to add indexes to fields like supplierId, warehouseId, and vesselId in the Orders collection. These indexes improve query performance when:
    - Filtering orders based on a specific supplier, warehouse, or vessel.
    - Performing aggregation lookups ($lookup) to join related collections.
    - Scaling queries as the number of documents in the Orders collection grows.

Constraints:
To ensure data consistency, validation is added to the status field in the Orders

collection. The status field is restricted to the following allowed values: ["pending", "ready", "inbound", "stock", "cancelled", "delivered"].

This is enforced using MongoDB's JSON Schema Validation, which ensures that only valid statuses can be inserted or updated in the collection.

3.3. Description of the CRUD application for the document database – security – registration / login, transactions, etc.

MongoDB is used as the document database to manage entities such as Orders, Suppliers, Agents, Warehouses, Vessels, Owners, Boxes, and Dispatches. Unlike relational databases, MongoDB allows for flexible, schema-less data storage using collections and documents.

The repository pattern is implemented to abstract the MongoDB operations, ensuring maintainability and separation of concerns. The application uses AutoMapper for data transformation between domain models and MongoDB collection entities.

Entity Generation

In MongoDB, data is stored as documents within collections. Each document is represented as a collection class, mapped to MongoDB's BSON format. For example, the BoxCollection class defines the structure of a document in the Boxes collection:

```
public class BoxCollection
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; } = null!;

    public int Length { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
    public double Weight { get; set; }

    public string OrderId { get; set; } = null!;
}
```

BsonId: Specifies that the Id property is the document's primary key.

BsonRepresentation: Ensures that the Id is stored as an ObjectId in MongoDB but exposed as a string in the application.
Relationships: References to other documents (like OrderId) are stored as string values to maintain relationships.

Create Operation
Documents are added to their respective collections using the InsertOneAsync method. Before insertion, validation ensures referential integrity.
Example: Creating a box document.

```
public async Task<Box> CreateBoxAsync(Box box)
{
   var boxCollection = mapper.Map<BoxCollection>(box);

   await collection.InsertOneAsync(boxCollection);
   box.Id = boxCollection.Id;

   return box;
}
```

Get Operation
Documents are queried using filters that specify conditions for matching fields. MongoDB's query filters are defined separately using the Builders<T>.Filter class. This design allows filters to be reusable and modular across various operations like querying, counting, or deleting.
The Find method is responsible for executing the query using the filter. Once the documents are fetched, they are mapped to the domain model for application use.

Example: Fetching all boxes for a given order:

```
public async Task<List<Box>> GetBoxesByOrderIdAsync(string
orderId)
{
// Step 1: Create a filter to match documents where OrderId equals
the provided orderId
   var filter = Builders<BoxCollection>.Filter.Eq(b => b.OrderId,
orderId);

// Step 2: Use the Find method to execute the query with the
```

```
filter

   var boxCollections = await
collection.Find(filter).ToListAsync();
// Step 3: Map the MongoDB collection results to the domain model
   return mapper.Map<List<Box>>(boxCollections);
}
```

Update Operation:
The method updates a list of boxes for a specific OrderId. Each box is validated to ensure it belongs to the correct order, and the documents are safely replaced without inserting new ones.

```
public async Task UpdateBoxesAsync(string orderId, List<Box>
boxes)
{
    // Map the input boxes to the MongoDB collection entity
    var boxEntities = mapper.Map<List<BoxCollection>>(boxes);

    foreach (var box in boxEntities)
    {
        // Ensure the correct OrderId is assigned
        box.OrderId = orderId;

        // Define a filter to match the specific box by its Id and
OrderId
        var filter = Builders<BoxCollection>.Filter.And(
            Builders<BoxCollection>.Filter.Eq(b => b.Id, box.Id),
            Builders<BoxCollection>.Filter.Eq(b => b.OrderId,
orderId)
        );

        // Replace the document if it exists; no insertion if not
found
        await collection.ReplaceOneAsync(filter, box, new
ReplaceOptions { IsUpsert = false });
    }
}
```

Delete Operation:

The delete operation is quite similar to get operation starting with creating a filter to match the box by its id and OrderId. Afterwards you call DeleteOneAsync to delete the document.

4. Graph database

4.1. Intro to graph databases

Graph databases are a type of NoSQL database designed to handle and represent data with complex relationships. Unlike relational databases, which are structured with tables, rows, and columns, graph databases use nodes, edges, and properties to store and represent data. This structure makes graph databases particularly well-suited for applications that require managing and querying complex connections.

In a graph database a node represents the program's entities or objects, such as users, products or locations. The edges show all the relations between the nodes, this could be friends, purchased or located in. As for the properties in a graph database, it stores the information about the nodes and edges, this could be a username or the date of a purchase.

This model allows for a more intuitive and efficient querying of relationships and patterns within the data. For example, finding the shortest path between two nodes or identifying clusters of interconnected nodes, which then can be done more naturally and efficiently in a graph database compared to a relational database.

One of the most commonly used graph databases—and the one we have implemented in our application—is Neo4j. Neo4j uses the Cypher query language, which is designed specifically for querying graph data. Other examples of graph databases include Amazon Neptune, OrientDB, and Azure Cosmos DB.

Graph databases offer a larger flexibility than relational databases, by accommodating changes in the data models without requiring schema modification. This enhances the querie’s performance, by more efficiently handling complex relationships, as well as giving a higher level of intuitiveness in providing a more natural way to both model and query connected data.

4.2. Database design + features

In a graph database, the data is represented as nodes and relationships (which also is known as edges).
The nodes are the fundamental units that represent the program's entities or objects, such as users or products. Each node can have labels that categorize it into different types, in the case of SpareHub this could be Vessels and Ports. The relationships connect nodes and represent the relation they have to each other, as well as showing directions, indicating the nature of the nodes connection. In SpareHub these relations would be between Box, Order and Dispatch, as well as the relations between owner, vessel and port. One of the relations in SpareHub would be a Vessel that is owned by an Owner, and that is docked at a Port.

In graph databases there are also properties, which are key-value pairs, used to store information about nodes and relationships. For the node Port these properties could be name and address. Whereas for a relationship like Vessels and Ports the properties can be since and until, describing how long a Vessel will stay at a certain port.

Indexing is crucial for improving query performance in graph databases, just like it is for relational databases. These indexes can be created on node properties to speed up the retrieval of the node based on specific criteria. This could be by indexing the OrderNumber in an Order, which allows fast lookups of Orders by OrderNumber.

In graph databases the query language is essential for interacting with the data. In Neo4j the query language used is Cypher, and allows pattern matching, filtering and manipulation of graph data using a SQL-like syntax. An example of a Cypher query could be:

```
MATCH (o:Owner)
RETURN o.id as id, o.name as name
```

Graph databases are specifically designed to handle large-scale and complex data efficiently. This happens by using Horizontal Scalability, Efficient Traversals and Caching. Many graph databases including Neo4j support horizontal scalability through clustering and sharding. This allows the database to distribute data across multiple servers, improving the performance and fault tolerance. Graph databases are optimized for traversing relationships, making them highly efficient for queries that involve more complex connections. This is especially beneficial for applications like social media, where relationship traversal is common. By using advanced

caching mechanisms and query optimization techniques, the performance of the program can be further enhanced.

4.3. Description of the CRUD application for the document database – security – registration / login, transactions, etc.

For our application we have created repositories containing CRUD operations that seamlessly interact with the rest of the backend, without having to change the code, when changing the database in use. The operations revolve around showing the relations between Owner, Vessel and Port.

For Owner, Vessel and Port, nodes are created describing the content and properties of each object:

```
public class VesselNode
{
    public required int Id { get; init; }

    public int OwnerId { get; init; }

    public required string Name { get; init; }
    public string? ImoNumber { get; init; }
    public string? Flag { get; init; }

    public required OwnerNode Owner { get; init; }

    [JsonIgnore]
    public ICollection<VesselAtPortRelationship> VesselAtPorts {
get; set; } = new List<VesselAtPortRelationship>();
}
```

Then an edge is set up for the Vessels docked at a Port, ensuring the right nodes of the relationship is specified:

```
public class VesselAtPortRelationship
{
    public required int VesselId { get; init; }
    public required int PortId { get; init; }
    public DateTime? ArrivalDate { get; set; }
    public DateTime? DepartureDate { get; set; }
```

```
    [JsonIgnore]
    public VesselNode VesselNode { get; set; } = null!;
    public PortNode PortNode { get; set; } = null!;
}
```

Repositories containing Cypher query is then set up with the wanted CRUD functionalities:

Get operations:
This operation fetches one or more objects, based on the criteria given in the query. An example of this would be fetching a Vessel by Id:

```
public async Task<Vessel> GetVesselByIdAsync(string vesselId)
    {
        await using var session = driver.AsyncSession();

        var query = @"
            MATCH (v:Vessel)-[:OWNED_BY]->(o:Owner)
            WHERE v.id = $vesselId
            RETURN v.id as id, v.name as name, v.imoNumber as
imoNumber,
                   v.flag as flag, o.id as ownerId, o.name as
ownerName";

        var result = await session.RunAsync(query, new { vesselId
});
        var record = await result.SingleAsync();

        return new Vessel
        {
            Id = record["id"].As<string>(),
            Name = record["name"].As<string>(),
            ImoNumber = record["imoNumber"].As<string>(),
            Flag = record["flag"].As<string>(),
            Owner = new Owner
            {
                Id = record["ownerId"].As<string>(),
```

```
                Name = record["ownerName"].As<string>()
            }
        };
    }
```

Create operations work by inserting documented data into the queries, and verifying the data before sending it. An example of that would be creating a Vessel:

```
public async Task<Vessel> CreateVesselAsync(Vessel vessel)
    {
        await using var session = driver.AsyncSession();

        if (string.IsNullOrEmpty(vessel.Id))
        {
            vessel = new Vessel
            {
                Id = Guid.NewGuid().ToString(),
                Name = vessel.Name,
                ImoNumber = vessel.ImoNumber,
                Flag = vessel.Flag,
                Owner = vessel.Owner
            };
        }

        var query = @"
            MATCH (o:Owner {id: $ownerId})
            CREATE (v:Vessel {
                id: $id,
                name: $name,
                imoNumber: $imoNumber,
                flag: $flag
            })-[:OWNED_BY]->(o)
            RETURN v.id as id, v.name as name, v.imoNumber as
imoNumber,
                   v.flag as flag, o.id as ownerId, o.name as
ownerName";

        var parameters = new
```

```
        {
            id = vessel.Id,
            name = vessel.Name,
            imoNumber = vessel.ImoNumber,
            flag = vessel.Flag,
            ownerId = vessel.Owner.Id
        };

        var result = await session.RunAsync(query, parameters);
        var record = await result.SingleAsync();

        return new Vessel
        {
            Id = record["id"].As<string>(),
            Name = record["name"].As<string>(),
            ImoNumber = record["imoNumber"].As<string>(),
            Flag = record["flag"].As<string>(),
            Owner = new Owner
            {
                Id = record["ownerId"].As<string>(),
                Name = record["ownerName"].As<string>()
            }
        };
    }
```

Update operations work by verifying the existence of the desired object of change, and thereafter inserting the new documented data into the object, overwriting the old. During update operation index of properties are not changed, even when the value of the property is changed. An example would be updating a Port:

```
public async Task UpdatePortAsync(string portId, Port port)
    {
        await using var session = driver.AsyncSession();

        var checkQuery = @"
            MATCH (p:Port {id: $portId})
            RETURN p";
```

```
        var checkResult = await session.RunAsync(checkQuery, new {
portId });
        if (!await checkResult.FetchAsync())
        {
            throw new NotFoundException($"Port with id '{portId}'
not found");
        }

        var query = @"
            MATCH (p:Port {id: $portId})
            SET p.name = $name,
            RETURN p.id as id";

        var parameters = new
        {
            portId = port.Id,
            name = port.Name
        };

        await session.RunAsync(query, parameters);
    }
```

Delete operations are similar to Update operations, because it also confirms the existence of the wanted object before deleting it. An example would be deleting an Owner:

```
public async Task DeleteOwnerAsync(string ownerId)
    {
        await using var session = driver.AsyncSession();

        var checkQuery = @"
            MATCH (o:Owner {id: $ownerId})
            RETURN o";

        var checkResult = await session.RunAsync(checkQuery, new {
ownerId });
        if (!await checkResult.FetchAsync())
        {
```

```
            throw new NotFoundException($"Owner with id '{ownerId}'
not found");
        }

        var query = @"
            MATCH (o:Owner {id: $ownerId})
            DELETE o";

        await session.RunAsync(query, new { ownerId });
    }
```

5. Conclusions (discussing the similarities and differences between used database types)

The system is primarily built around MySQL as the relational database for managing core entities such as Orders, Suppliers, Warehouses, Vessels, and Agents. MySQL ensures strong referential integrity, structured relationships, and efficient handling of complex queries through joins, which is ideal for data requiring strict consistency and predefined relationships.

However, for certain parts of the system, particularly the Boxes table, a document-based database like MongoDB can offer notable advantages. MongoDB's flexibility allows for dynamic schema changes, which simplifies handling boxes with attributes like dimensions and weights. This flexibility eliminates the need for schema modifications when the box structure evolves, which would otherwise require careful adjustments in MySQL.

Additionally, MongoDB performs well in scenarios where the amount of data associated with a single order grows over time. Instead of relying on expensive join operations, querying boxes by their OrderId is straightforward and efficient using indexed filters. By storing box data as individual documents, MongoDB reduces complexity and enhances performance for read-heavy operations.

Graph databases offer a unique and powerful approach to managing and querying data with complex relationships. It uses nodes, edges and properties to represent its data, ideal for applications with interconnected data. Depending on the type of graph database a specialized query language like Cypher is used to traverse and query the graph data. In common among graph databases, horizontal scalability is used to make clustering and sharding work more seamless, it also excels in the performance of queries involving complex relationships. A graph database hereby provides a more flexible, efficient and intuitive way of managing queries, making it easier to handle more complex data with higher efficiency. In our project this revolves around our Vessels, Owners and Ports, along with Orders and Dispatches, showing the connection between the nodes and their relation to each other.

In conclusion, while MySQL remains the primary database for the system due to its relational nature and enforcement of data integrity, MongoDB presents clear benefits for managing Boxes, where flexibility and scalability are more critical. A hybrid approach that leverages both databases ensures the system maintains robust consistency for core data while efficiently handling high-volume, evolving data like boxes.