**Database Design Assignment**

**Deliverable: -** **Physical Entity Relationship diagram of database.**

An Entity Representation Diagram is a blueprint for creating the foundations of any application that uses information stored in the databases. Data modelers us the ER diagrams like the architects use their drawing boards.

Any application that uses a database should have in its design documentation an ER diagram that reflects the structure of the stored data.

ER diagram is the graphical depiction of relationships between all the entities involved in the system. Its major components are Entities, Attributes, and Relationships.

ER diagram means Entity Relationships diagram. Entity means object of system, generally we refer entity as database table, the ER diagram represent the relationship between each table of database. ER diagram represent entity with attributes is a property of entity. If we assume entity is a database table, then all the columns of the table are treated as attributes.

**Step by Step construction of an ER diagram**

The process starts by sketching a conceptual model, then converting it to a logical model, and finally to the physical model.

**Identifying Entities**

**Step 1**: **Building a Conceptual Data Model for the Application.**

The first thing to do is to identify the entities involved in the system and their attributes. The following entities can be found:

Entities: Types of Information

Order Item

Category

Cart

Order

Product

User Login

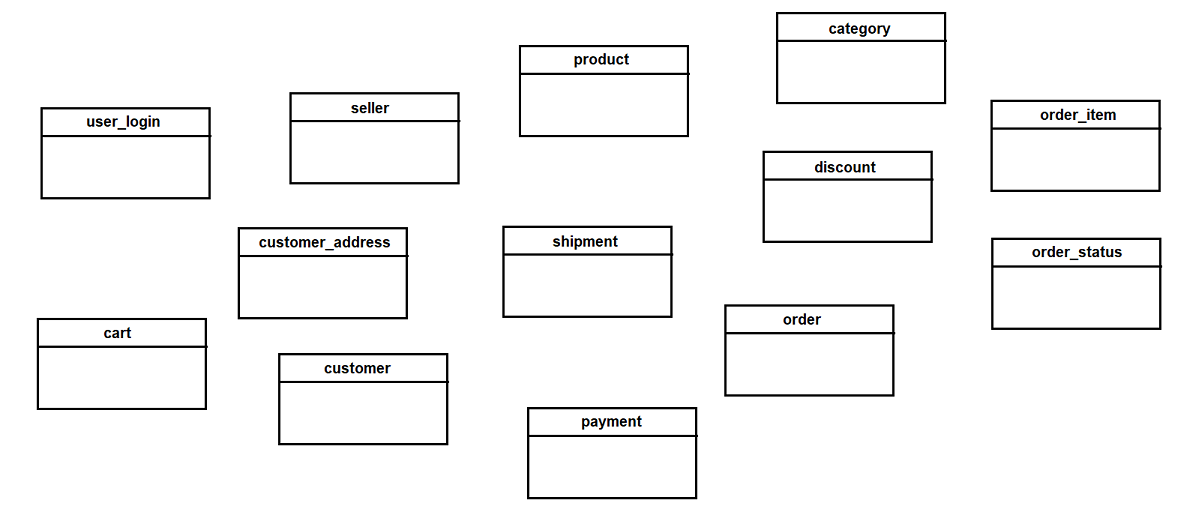
Customer

Seller

* **Seller:** The entity represents the sellers who creates an account to sell products to the customers on the ecommerce platform.
* **Customer:** This entity represents the customers who create an account to place orders on the ecommerce platform.
* **Product:** Represents the set of products available for purchase on the platform.
* **Category:** Categories in which the products are grouped.
* **Order:** Product orders placed by customers.
* **Order\_item:** Each item that is part of an order.
* **Cart:** The customer’s virtual basket or shopping cart, which stores items before they are purchased and become part of an order.

**Identifying Relationships**

To build ER diagram for the e-commerce application, start by building the entities without relationships or attributes.



**Relationships**

The relationship is the connection between the entities. How one entity is related to the other entity in the e-commerce application. For example, the seller adds the products, the customer can place several orders, an order can contain one or several item, each of which represents a single product.

Relationships are represented as lines between entities. Relationships, like attributes, also have additional information that describes the cardinality of each relationship.

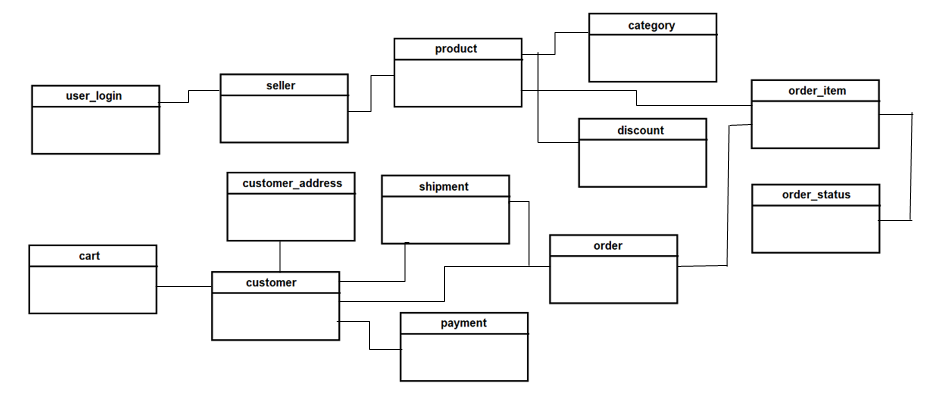
There are two symbols for cardinality:

* A straight line represents “one’.
* A “Crow’s Foot” or fork represents “many”.

The relationship between the entities are as follows:

* A seller/ vendor can add products to the e-commerce application. Therefore, between **Seller** and **Product** there must be a **one-to-many** relationship.
* A customer can place several orders. Therefore, between **Customer** and **Order** there must be a **one-to-many** relationship.
* An order can contain one or several items, each of which represents a single product. **Order\_Item** is a dependent entity of **Order**, since it has no reason to exist if an order does not exist. In addition, **Order\_Item** is related to **Product** through a **one-to-many** relationship: each **Order\_Item** is related to one **Product**, and a **Product** can be related to multiple **Oder\_Item**.
* An order is associated with one payment and one shipment, but each payment and each shipment can include multiple orders. For this reason, there are one-to-many relationships between **Payment** and **Order** and between **Shipment** and **Order**.
* A product can belong to a single category: there is a **one-to-many** relationship between **Product** and Category.
* The shopping cart is a dependent entity of **Customer**, so **Cart** maintain a dependency relationship with **Customer**. In turn, each instance of **Cart** is related to a product, so the entity has **many-to-one** relationship with **Product.**

Having identified all the entities and the relationships between them, we can draw the conceptual model.



**Step 2: Building the Logical Model for the application.**

After identifying the entities and relationships that compose the ecommerce application and sketching the conceptual model, we need to define the attributes that compose each entity. Once we add the attributes, the logical diagram will be completed.

It is important to highlight that all the entities mut have a primary identifier.

Entity Construction

Let’s define the attributes for each entity.

**Seller entity**

|  |  |
| --- | --- |
| seller\_id | integer |
| seller\_name | string |
| address | string |

**Customer Entity**

|  |  |
| --- | --- |
| customer\_id | integer |
| first\_name | string |
| last\_name | string |
| email | string |
| password | string |
| address | string |
| phone\_number | string |

Cart is dependent entity of Customer. Cart includes each product added to the shopping cart and the quantity of the product.

**Cart**

|  |  |
| --- | --- |
| cart\_id | integer |
| Quantity | integer |

Product entity has attributes to store Product SKUs, price, discount, specification and image for each product.

**Product**

|  |  |
| --- | --- |
| product\_id | integer |
| product\_image | url |
| price | decimal |
| discounts | decimal |
| specification | string |
| SKU | string |

**Category** Entity

|  |  |
| --- | --- |
| category\_id | integer |
| category\_name | string |

The Order entity stores the date of each order and its total price.

**Order**

|  |  |
| --- | --- |
| order\_id | integer |
| order\_date | date/time |
| total\_price | decimal |

Order\_Item is the dependent entity of Order, it stores te quantity and price of each item included in an order.

**Order\_Item**

|  |  |
| --- | --- |
| order\_item\_id | integer |
| quantity | integer |
| price | decimal |

**Order\_status**

|  |  |
| --- | --- |
| ready\_to\_ship | string |
| packed | string |
| delivered | string |

The Payment entity stores the date of payment, the means of payment and the amount paid.

**Payment**

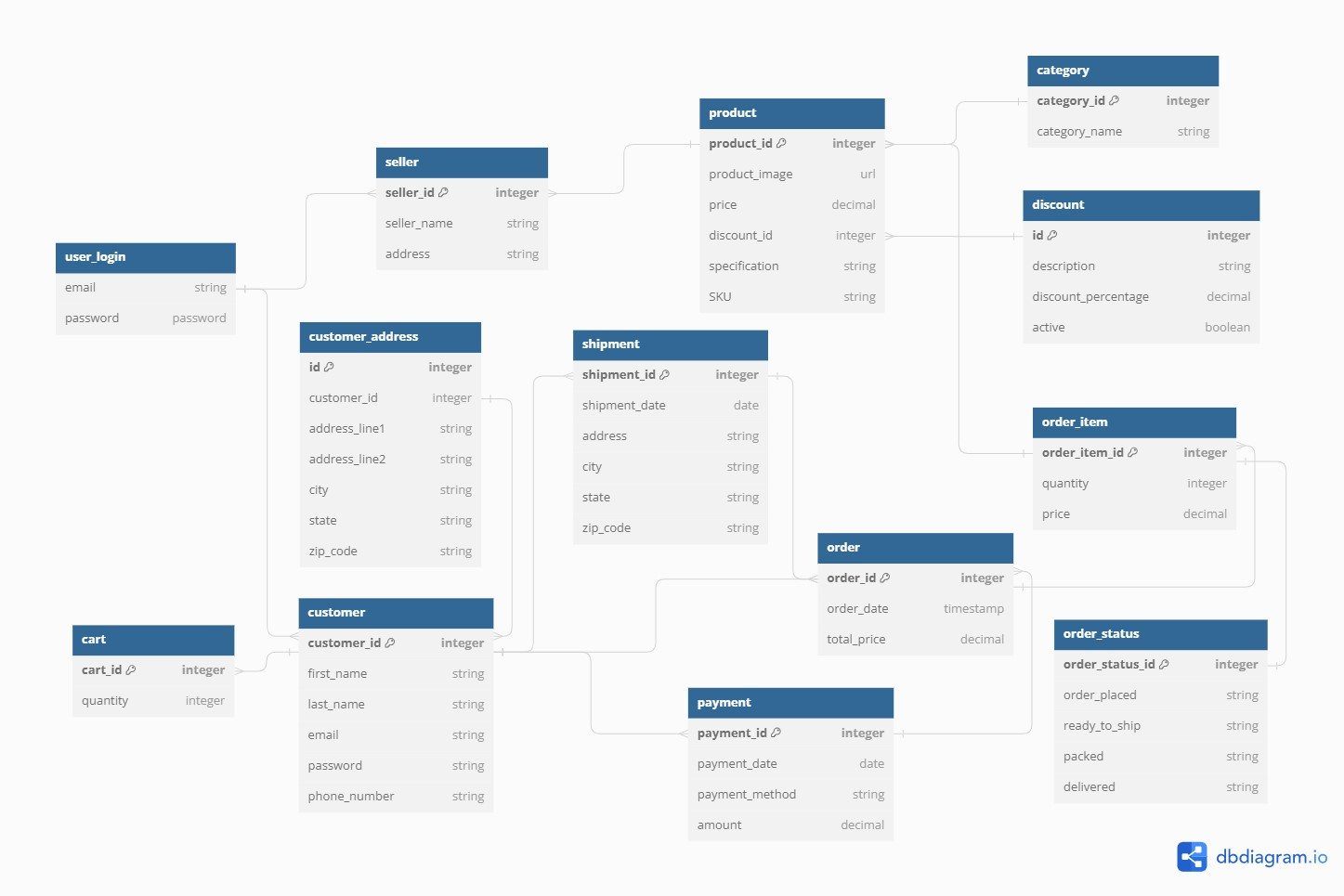
|  |  |
| --- | --- |
| payment\_id | integer |
| payment\_date | date |
| payment\_method | string |
| amount | decimal |

And finally, Shipment stores the shipping date, address, city, state, country, and postal code.

**Shipment**

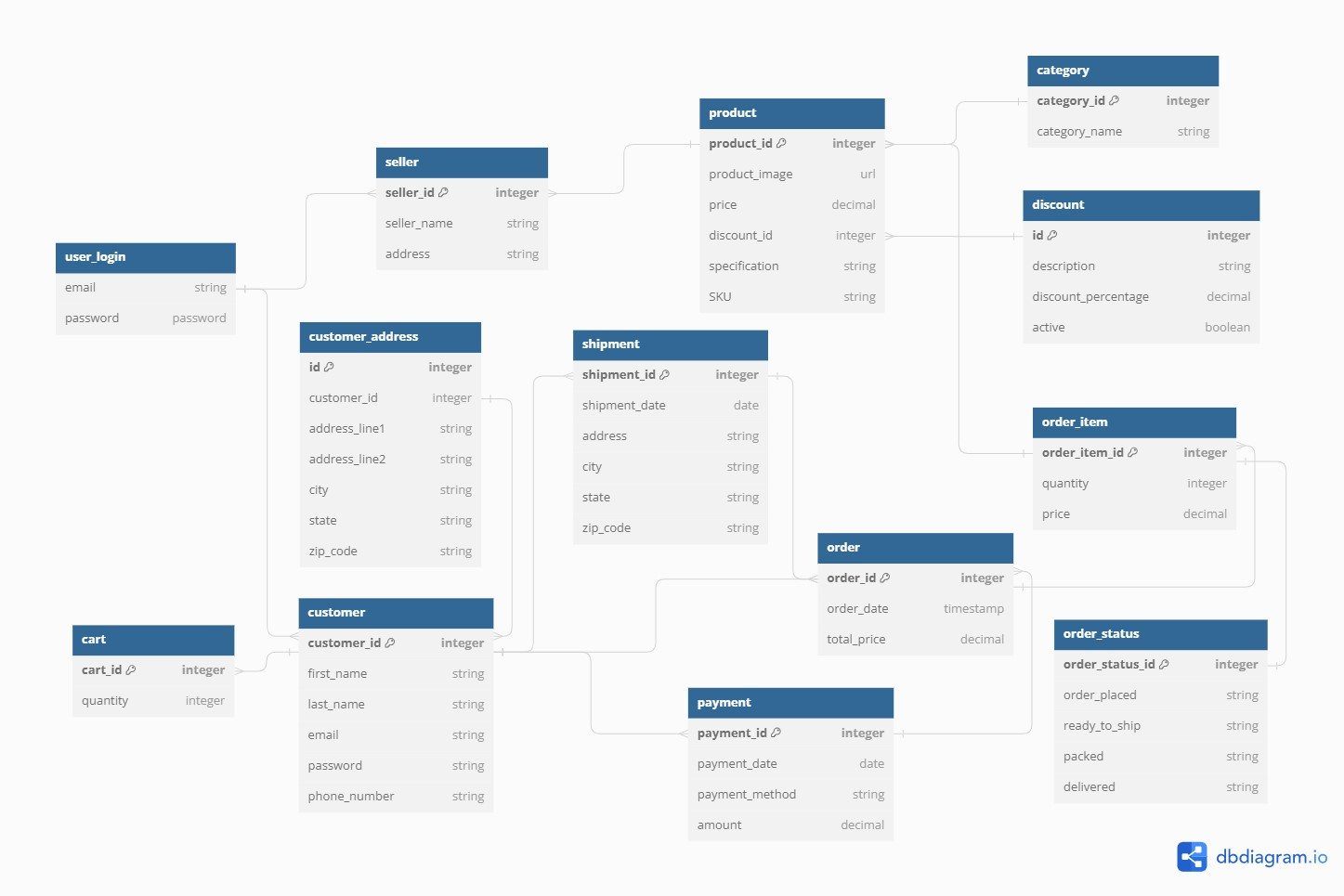
|  |  |
| --- | --- |
| shipment\_id | integer |
| shipment\_date | date |
| address | string |
| city | string |
| state | string |
| country | string |
| zip\_code | string |

Once we have added the attributes to the entities, the logical model will be completed.



**Step 3: Create the Physical Model**

The entities contain the Primary key and the Foreign key.



**Designing a User Authentication Module**

The design of an application should specify mechanisms to prevent unauthorized users from entering it. Through authentication, an application can verify that each user attempting to log in is who he/she claims to be and has permission to access and use the application.

Requirements of User Authentication Module

To provide protection against unauthorized login attempts, a user authentication module must provide the following capabilities as the functional requirements:

* Register new users.
* Send confirmation emails.
* Provide secure options for recovering a forgotten password.
* Protect authentication data from unauthorized access.
* Support authentication via third-party services.
* Define roles and permission sets per role.

Building a Database Model for User Authentication

The design of an authentication module needs to include a table that stores the information for validating each user’s login. This table contains the email and the password hash.

Storing encrypted passwords is not enough as a security measure. Encryption functions are bijective functions; that is, every encryption function has an inverse function for decrypting the data. This means there is always a risk that an encrypted key can be decrypted. For this reason, password should not be stored in the database, even in the encrypted form.

Hashing functions are not bijective. It is impossible to obtain a password from its hash. However, hashing can be used to validate the password.: if the password entered by the user renders the same hash as the one stored in the table, the hashing algorithm guarantees the password is same as the one set by the user. Th password validation algorithm compares the hash of the password entered by the user with the hash stored in the user table for the given email address.

**Password Recovery**

An effective user authentication system provides an option to recover a forgotten password. The best practice is to provide the user with a link on the login page to manage password recovery. This is done by requesting a user ID (email address), validating that the user ID corresponds to an active user account, and sending the email with a randomly generated token to the previously validated email address.

The recovery token must be short-lived since its mere existence implies a security risk. To implement a password recovery mechanism, we must add two more fields to the authentication data table: one for the recovery token and another to capture the timestamp with the date and time the token was generated so that it can be deleted after a set period.

**Multi-Factor Authentication**

Separating the concepts of the user identity and the user account is a necessary step in implementing multi-factor authentication (MFA) systems. MFA is best practice for strengthening a user authentication system. It is based on the parallel use of validation elements (factors) of different types. For example, it may be a piece of information the user knows (a password or a set of personal data, such as nationality, year of birth, etc.) used in parallel with a physical element the user possesses (a phone or other device). Each authentication factor requires the data stored in user tables to validate the information provided by the user at login.



**Deliverable: - Selection of the database.**

The challenges for e-commerce database, A well configured database system should:

* Guarantee data availability 24/7,
* Maintain a high polling rate during periods of increased usage,
* Save large amounts of data,
* Provide information about changes (example: availability of given products from the product catalog) dynamically and an ongoing basis.
* Orders might have multiple products (and their variants)
* Orders might have different states that might change multiple times a day.
* Orders might be assigned to customers.
* Customers might be user with login name, password, address etc.
* During peak sale periods , such as Black Friday or Cyber Monday, that translates into an increased number of queries.

For this reason, e-commerce companies should focus on database scalability.

Just as products are physically kept in warehouses, in the virtual world information about them is stored in databases.

Now let’s determine the type of Database. We must consider the two main database types: Relational Databases (SQL) and Non-Relational Databases (NoSQL).

When choosing a database management system (DBMS) for the online shopping store, we need to pay attention to different aspects: flexibility, high availability, reliability, handling multiple inquire and data timelines.

SQL is a structured Query Language, a language for retrieving data from a relational databae. Relational Database Management System is like spreadsheets and uses tables, columns, and rows to organize and retrieve data. It is built using the standard query language (SQL) and all the data is related to each other. Examples include MySQL, PostgreSQL, MariaDB, Microsoft SQL, Amazon RDS, and Azure SQL Database.

NoSQL Database

A non-relational database is a non-tabular database with a flexible schema that works well for storing unstructured data. A non-relational database can store related data. It does so by nesting related data with single data structure instead of splitting them between tables.

There are various NoSQL databases, and the most popular ones are document stores and key stores. Examples include MongoDB, Apache Cassandra, Amazon DynamoDB, Azure Cosmos DB and Couchbase.

A relational database will provide a simple and robust platform to create the database while NoSQL offers better data flexibility, scalability, and slightly better performance.

There are 5 basic differences between SQL and NoSQL:

|  |  |
| --- | --- |
| SQL | NoSQL |
| Clearly defined Data relationships. | No relationship, the data in our database is loosely coupled. |
| Data is stored in Tables | Data is stored in Documents, Graphs as the so-called key-value |
| Defined Schema | Dynamic Schema, unordered data |
| Preferred in the case of multi-line operations | Preferred when the speed of the data acquisition is important |
| Vertically Scalable | Horizontally Scalable |

The NoSQL databases perfectly match the requirements and needs of the e-commerce application in term of data availability and storage. The most popular database system of NoSQL type is MongoDB.

What is MongoDB?

With MongoDB (and any other NoSQL database) we can build safe, reliable, scalable, cloud-based database.

MongoDB is a document database for easy creation and scaling. Documents are created and stored in BSON (Binary JSON) format, which means its very simple to convert queries and results into a format that frontend code understands. This removes the need to write object-relation mappers (ORMs). MongoDB scales extremely well, its designed to be distributed across multiple machines. NoSQL databases do not come with enforced schemas. Introducing new field is fast.

The NoSQL solution includes hierarchy, automatic fragmentation, and bult-in replication for better scalability and high availability.

When we work with MongoDB, we create JSON-based entries called “documents” in the database and group them in so-called “collections”.

**Advantages of NoSQL database in e-commerce application based on MongoDB:**

* Dynamic Schema

They do not have the rigid tabular row-and-column data model like traditional relational databases. The documents in the collection do not have the same fields, and a given field can have different types depending on the document. This increases the flexibility of mapping to entities or object. Structure of the document inside the collection is similar.

* Easy hierarchization of data

The use of JSON format makes it easy to structure the data. We can do this by embedding one document in another or by providing references. The use of a given method should be considered individually for each collection. Embedding is recommended because it allows to obtain data with a single query, which improves the systems performance. References are considered for complex hierarchy representations or when the benefits of embedding do not outweigh the effects of data duplication.

* Replication

MongoDB utilizes a concept called Replica Set, which is a set of nodes containing the same data. This enables data replication, the purpose of which is to increase availability and protect against database server failures. A properly designed architecture allows faster access to data.

The replica set consists of one node, called the primary member and secondary member. There is a special member of such a set, the Arbiter, which does not contain a copy of the data but is used to select an alternative if the main server is unavailable. Saving operations are performed only on the Primary instance, from which the built-in MongoDB mechanism then copies the data to the other instances.

In clocking mechanism each of the nodes poll the others every 2 seconds to check their availability. If the main server is unavailable, a new one is selected.

Deploy replica set

This process consist of selecting the one with the highest priority from the remaining instances. According to documentation, the replica can have up to 50 nodes, of which 7 can participate in the selection process (voting), the successor is chosen from among them. Other servers, named Non-voting members, must have the properties votes and priority set to 0. Setting an uneven number of voting is recommended, hence the minimum number of nodes in a replica set is 3.

MongoDB schema design has two choices for every piece of data. Either embed that data directly or reference another piece of data using the $lookup operator (similar to a JOIN)

Collections: A collection in MongoDB is similar to a table in a relational database. Documents within a collection can have different fields and structures, allowing us to store data with varying shapes.

**Deliverable: - Type of Data Modelling.**

**MongoDB provides two types of data model: - Embedded data model and Normalized data model.**

**Embedding Documents**

One of the most powerful features of MongoDB is the ability to embed documents within other documents. This allows you to store related data together, which can improve query performance by reducing the need for joins. When designing your schema, consider embedding documents when:

* The relationship between entities is "contains" or "one-to-many."
* You need to retrieve the entire related data set at once.
* The related data set is small and unlikely to grow large.

For example, if you are designing a schema for a blog application, you might embed comments within a post document:

{

"\_id": ObjectId("507f191e810c19729de860ea"),

"title": "MongoDB Schema Design Best Practices",

"content": "...",

"author": "John Doe",

"comments": [

{

"author": "Jane Smith",

"content": "Great post!",

"timestamp": ISODate("2023-04-07T12:34:56.789Z")

},

{

"author": "Alice Brown",

"content": "Very informative.",

"timestamp": ISODate("2023-04-07T13:14:22.123Z")

}

]

}

**Use References**

In some cases, embedding documents may not be the best solution. For example, if the related data set is large or frequently updated, embedding documents can lead to performance issues and increased storage costs. In such situations, it's better to use references to link related documents.

Consider using references when:

* The relationship between entities is "one-to-many" or "many-to-many."
* You need to retrieve only a subset of the related data set.
* The related data set is large or frequently updated.

For example, in an e-commerce application, you might use references to link orders with customers:

// customer document

{

"\_id": ObjectId("507f1f77bcf86cd799439011"),

"name": "Alice",

"email": "alice@example.com"

}

// order document

{

"\_id": ObjectId("507f191e810c19729de860ea"),

"customerId": ObjectId("507f1f77bcf86cd799439011"),

"items": [

{ "productId": "1001", "quantity": 2 },

{ "productId": "1002", "quantity": 1 }

],

"total": 59.98

}

To retrieve the customer information for an order, you can use a query like this:

db.customers.findOne({ "\_id": order.customerId });

**Deliverable: - Handle Searching in the E-commerce Application.**

**How to implement Searching?**

The limitation that MongoDB has on the full text search feature is that it indexes documents on the word level, so it’s impossible by using a text index to do what it’s called partial matching. This is, matching partial parts of a word.

Therefore, a more powerful text indexing platform is useful. **Elastic Search**, as it provides out of the box a full set of RESTful API endpoints that makes it very easy to test.

**What is Elastic Search?**

Elasticsearch is a free and open source, document-oriented, schema-free, RESTful, distributed search and analytics engine built on top of Apache Lucene and developed by Elastic. Lucene is the most popular Java-based full text search index implementation. It started as a scalable version of the Lucene open-source search framework then added the ability to horizontally scale Lucene indices. It extends Lucene’s indexing and search functionalities using RESTful APIs, and it achieves the distribution of data on multiple servers using the index and shards concept. Elasticsearch is based on JSON and is suitable for search use cases against time series data, structured or unstructured text, numerical data, or geospatial data.

Deployment options include self-hosting Elasticsearch, where the user is self-managing their instance, and Elastic’s cloud-hosted variant included in Elastic Cloud (which includes the rest of the ELK stack).

Elasticsearch allows you to store, search, and analyse huge volumes of data quickly and in near real-time and give back answers in milliseconds. It’s able to achieve fast search responses because instead of searching the text directly, it searches an index. It uses a structure based on documents instead of tables and schemas and comes with extensive REST APIs for storing and searching the data. At its core, Elasticsearch is as a server that can process JSON requests and give you back JSON data.

**Why Elasticsearch? Why not database-search?**

Elasticsearch is the obvious choice if you have a sizable dataset, want a wide range of query capabilities, such as a fuzzy search or searches where the user can select several filters on the data, or you want to perform some predictive analysis on the data. If you have thousands to millions rows of dataset, and you have hundreds of transaction per second, also various customized parameters which can be a factor for search, relational database cannot do the heavy lifting. Elasticsearch provides a wide range of features and customization options that allow you to fine-tune your search capabilities. From text analysis techniques like n-grams, synonyms, and stemmers, to filters for stop words and character processing, Elasticsearch offers a comprehensive toolkit for creating a powerful search index. By leveraging the full potential of Elasticsearch, you can unlock the ability to handle complex search queries, support autocomplete functionality, and provide intelligent recommendations.

1. Easy to scale.

Elasticsearch allows to start small but will grow with business. It is built to scale horizontally out of the box. As we need more capacity, we can add more nodes, and let the cluster recognize itself to take advantage of the extra hardware.

1. RESTful API

Elasticsearch is API driven. Almost any action can be performed using a simple RESTful API using JSON over HTTP. Responses are always in JSON, which is both machine and human readable.

1. Per-operation Persistence

Elasticsearch puts the data safety first. Document changes are recorded in transaction logs on multiple nodes in the cluster to minimize the chance of any data loss.

1. Excellent Query DSL

The REST API exposes a very complex and capable query DSL, that is very easy to use. Every query is just a JSON object that can practically contain any type of query, or even several of them combined. Using filtered queries, with some queries expressed as Lucene filters, help leverage caching and thus speed up common queries, or complex queries with parts that can be reused. Faceting, another very common search feature, is just something that upon-request is accompanied to search results, and then is ready for you to use.

1. Muti-tenancy

We can host multiple indexes on one Elasticsearch installation – node or cluster. Each index can have multiple “types”, which are essentially completely different indexes. We can query multiple types and multiple indexes with one simple query.

1. Support for advanced search features (Full Text)

Elasticsearch uses Lucene under the covers to provide the most powerful **full text search** capabilities available in any open-source product. Search comes with multi-language support, a powerful query language, support for geolocation, context aware did-you-mean suggestions, **autocomplete** and search snippets.

1. Document Oriented

Store complex real-world entities in Elasticsearch as structured JSON documents. All fields are indexed by default, and all the indices can be used in single query, to return results at breath taking speed.

1. Schem free

Elasticsearch allows you to get started easily. Send it a JSON document and it will try to detect the data structure, index the data and make it searchable.

**How does Elasticsearch work?**

Fundamentally, Elasticsearch organizes data into documents, which are JSON-based units of information representing entities. Documents are grouped into indices, like database, based on their characteristics. Elasticsearch uses inverted indices, a data structure that maps words to their document locations, for an efficient search. Elasticsearch’s distributed architecture enables the rapid search and analysis of massive amounts of data with almost real-time performance.

Documents: Documents are basic unit of information that can be indexed in Elasticsearch expressed in JSON, which is the global internet data interchange format. Document is like a row in a relational database, representing a given entity. In Elasticsearch, a document can be more than just text, it can be any structured data encoded in JSON. The data can be things like numbers, strings, and dates. Each document has a unique ID and a given data type, which describes wat kind of entity the document is.

Indices: An index is a collection of documents that have similar characteristics. An index is the highest-level entity that can be queried against Elasticsearch. Index is like a database in a relational database schema. Any documents in an index re typically logically related. In an e-commerce application, we can have index for Customers, one for Products, one for Orders and so on. An index is identified by a nae that is used to refer to the index while performing indexing, search, update and delete operations against the documents in it.

Inverted Index: An index in Elasticsearch is called an inverted index, it is the mechanism by which all search engines work. It is a data structure that stores a mapping from content, such as words or numbers, to its locations in a document or a set of documents. Basically, it is a hashmap-like data structure that directs you from a word to a document. An inverted index doesn’t store strings directly and instead splits each document up to individual search terms (i.e. each word) then maps each search term to the documents those search terms occur within. This serves as a quick look-up of where to find search terms in a given document. By using distributed inverted indices, Elasticsearch quickly finds the best matches for full-text searches from even very large data sets.

**Backend components**

**Cluster:** An Elasticsearch cluster is a group of one or more node instances that are connected. The power of an Elasticsearch cluster lies in the distribution of tasks, searching, and indexing, across all the nodes in the cluster.

**Node:** A node is a single server that is a part of a cluster. A node stores data and participates in the cluster’s indexing and search capabilities. An Elasticsearch node can be configured in different ways:  
  
**Master Node** — Controls the Elasticsearch cluster and is responsible for all cluster-wide operations like creating/deleting an index and adding/removing nodes.  
  
**Data Node** — Stores data and executes data-related operations such as search and aggregation.  
  
**Client Node** — Forwards cluster requests to the master node and data-related requests to data nodes.

**Index**: An index is like a ‘database’ in a relational database. It has a mapping which defines multiple types. An index is a logical namespace which maps to one or more primary shards and can have zero or more replica shards.

**Type**: A type is like a ‘table’ in a relational database. Each type has a list of fields that can b specified for documents of that type. The mapping defines how each field in the document is analysed.

**Shards**

Elasticsearch provides the ability to subdivide the index into multiple pieces called shards. Each shard is in itself a fully functional and independent “index” that can be hosted on any node within a cluster. By distributing the documents in an index across multiple shards, and distributing those shards across multiple nodes, Elasticsearch can ensure redundancy, which both protects against hardware failures and increases query capacity as nodes are added to a cluster.

**Replicas**

Elasticsearch allows you to make one or more copies of your index’s shards which are called “replica shards” or just “replicas”. A replica shard is a copy of a primary shard. Each document in an index belongs to one primary shard. Replicas provide redundant copies of data to protect against hardware failure and increase capacity to serve read requests like searching or retrieving a document.

Once you have Elasticsearch installed, this is the overall process we’ll follow:

1. Create the index for our documents.
2. Import our MongoDB collection into ES with a tool called mongo-connector.
3. Migrate the index created by mongo-connector in ES to the index we created in step 1.
4. Try out our new index and see how documents are indexed all the time while we keep the mongo-connector running.

ES provides different analysers which serve as a starting point for creating custom analysers that suit better to any index needs. One of the alternatives provided by ES is called edge\_ngrams analyser. To understand what edge n-grams are, we first need to understand what n-grams are. **an n-gram is a contiguous sequence of n items from a given sequence of text or speech**. So, let’s say you have the word adidas, then the 1-grams or unigrams will be: [a, d, i, d, a, s].

Increasing n by 1, we get the bigrams of adidas: [ad, di, id, da, as]

Now we can see what edge n-grams are, and according to the ES documentation: **Edge n-grams are anchored to the beginning of the word.**

Which means that for adidas, the edge n-grams will be: [a, ad, adi, adid, adida, adidas]

If you have the word adidas indexed with its edge n-grams, you can easily create an autocomplete search module. Because if user types a, it will match, if the user types of ad it will match, if the user types ada it won’t match anymore and the autocomplete option would disappear.

So, this edge n-gram thing should be definitely part of our index, and this is how we’ll define it:

{  
 "filter": {  
 "autocomplete\_filter": {  
 "type": "edge\_ngram",  
 "min\_gram": 3,  
 "max\_gram": 20  
 }  
 }  
}

So, with this json object we’re defining a token filter (filter) called “autocomplete\_filter”. And we’re saying that it will be an edge\_ngram filter which will have from 3-grams up to 20-grams. The reason we used 3 as minimum is because for very big databases, having unigrams would slow down the performance a lot, since lots of documents would match the search. That’s why many websites that have autocomplete function ask users to type at least three characters until they can suggest alternatives.

Now that we have our token filter defined, we need to define our custom analyser:

{  
 "analyzer": {  
 "autocomplete": {  
 "type": "custom",  
 "tokenizer": "standard",  
 "filter": [  
 "lowercase",  
 "autocomplete\_filter"   
 ]  
 }  
 }  
}

Here we define a custom analyser called “autocomplete”, we tell ES that it will be a custom analyser, that will use the standard tokenizer and we set two filtering steps: lowercase and after that we set our custom autocomplete\_filter.

Now that we defined the filter and the analyser, let’s create the index. Execute the following curl command:

$ curl -H 'Content-Type: application/json' \  
 -X PUT http://localhost:9200/fulltext\_opt \  
 -d \  
 "{ \  
 \"settings\": { \  
 \"number\_of\_shards\": 1, \  
 \"analysis\": { \  
 \"filter\": { \  
 \"autocomplete\_filter\": { \  
 \"type\": \"edge\_ngram\", \  
 \"min\_gram\": 3, \  
 \"max\_gram\": 20 \  
 } \  
 }, \  
 \"analyzer\": { \  
 \"autocomplete\": { \  
 \"type\": \"custom\", \  
 \"tokenizer\": \"standard\", \  
 \"filter\": [ \  
 \"lowercase\", \  
 \"autocomplete\_filter\" \  
 ] \  
 } \  
 } \  
 } \  
 } \  
 }"  
**{**"acknowledged":true**}**

The fulltext\_opt in the endpoint URL tells ES to create a new index named like that. The reason I chose that name is because our MongoDB collection is named fulltext, and when we import it the first time to ES a fulltext index will be created automatically. We’ll later move all the documents from fulltext to the optimized fulltext\_opt index.

The last thing we have to do in our fulltext\_opt index is creating the mappings. Mappings are just groups of documents. We’ll create a mapping called articles and we’ll define the property title and content on it:

$ curl -H 'Content-Type: application/json' \  
 -X PUT http://localhost:9200/fulltext\_opt/\_mapping/articles \  
 -d \  
 "{ \  
 \"articles\": { \  
 \"properties\": { \  
 \"title\": { \  
 \"type\": \"string\", \  
 \"analyzer\": \"autocomplete\" \  
 }, \  
 \"content\": { \  
 \"type\": \"string\" \  
 } \  
 } \  
 } \  
 }"  
**{**"acknowledged":true**}**

The acknowledged: true response means our index was successfully created and the mappings added. Now it’s time to import the documents from our MongoDB into it.

2**. Importing from MongoDB into ES**

To import our documents, we could simply insert them manually into our ES index. The problem is that in real life we want to keep both MongoDB and our index synchronized, so that anytime a new document is inserted, the same document will be indexed in ES.

Fortunately for us, there’s a tool called mongo-connector that does what we need. And even better, it has support for Elastic Search. It will consume documents from our MongoDB and put them in our ES index.

The next step is to start MongoDB server as a replica set.To run MongoDB as a replica set just pass the --replSet option when starting it and give the replica set a name (rs0):

# Set the dbpath to wherever you have your data $ mongod --dbpath /data/mongodb/db/ --replSet rs0

You’ll probably see some message like this one:

016-07-30T16:17:45.881+0900 **[**rsStart] replSet can't get local.system.replset config from self or any seed (EMPTYCONFIG) 2016-07-30T16:17:45.881+0900 [rsStart] replSet info you may need to run replSetInitiate -- rs.initiate() in the shell -- if that is not already done

All you have to do is open the mongo shell, and run rs.initiate(). It’s possible that you might see this error message when trying to initiate the replica set:

$ rs.initiate()  
 {  
 "info2" : "no configuration explicitly specified -- making one",  
 "me" : "mbp-mauricio:27017",  
 "ok" : 0,  
 "errmsg" : "couldn't initiate : can't find self in the replset config"  
 }

The problem is that the replica can’t find the machine with name mbp-mauricio in this case. All you have to do is go to your /etc/hosts file and add an entry:

* + - 1. **[**your-machine-name]

MongoDB is up and running, now let’s start ES. Go into ES installation directory and run:

$ ./bin/elastic

All set, run the mongo-connector.

$ mongo-connector -m 127.0.0.1:27017 -t 127.0.0.1:9200 -d elastic2\_doc\_manager

So here we basically tell mongo-connector to consume MongoDB data from localhost:27017 and send it to the ES instance running on localhost:9200. All this will be done by using the elastic2\_doc\_manager. After a while (depending on how many MongoDB databases you have and how big they are), you should be able to see the new indexes in your ES instance. So if you call the corresponding ES endpoint to list indices, you should see this:

1. $ curl localhost:9200/\_cat/indices?v  
   health status index pri rep docs.count docs.deleted store.size pri.store.size  
   yellow open fulltext 5 1 2 0 10.9kb 10.9kb  
   yellow open fulltext\_opt 1 1 0 0 159b 159b

You might have more entries if you had other databases in your MongoDB instance. The good thing of mongo-connector is that it’s super configurable, so you can tell it which collections from which databases you want to import.

Moving documents between indices

So we have now two indices, one created by mongo-connector which is not optimized and has our two documents, and another one optimized but empty. All we have to do now is copy the documents between indices.

There is a great tool for this purpose called elasticdump which makes this task extremely easy. You can install it via NPM:

$ npm install -g elasticdump

With elasticdump you can import analysers, mappings and data from one ES index into another (or even into a json file). In our case we don’t care about analysers and mappings, we’ll just import the data since the analyser and mappings are already defined in our fulltext\_opt index.

$ elasticdump \  
 --input**=**http://localhost:9200/fulltext \  
 --output**=**http://localhost:9200/fulltext\_opt  
Mon, 01 Aug 2016 01:21:10 GMT | starting dump  
Mon, 01 Aug 2016 01:21:10 GMT | got 2 objects from source elasticsearch **(**offset: 0**)**  
Mon, 01 Aug 2016 01:21:10 GMT | sent 2 objects to destination elasticsearch, wrote 2  
Mon, 01 Aug 2016 01:21:10 GMT | got 0 objects from source elasticsearch **(**offset: 2**)**  
Mon, 01 Aug 2016 01:21:10 GMT | Total Writes: 2  
Mon, 01 Aug 2016 01:21:10 GMT | dump complete

Now if you run again the indices query in ES you should see that docs.count for the fulltext\_opt index has been changed to 2 instead of 0:

$ curl localhost:9200/\_cat/indices?v health status index pri rep docs.count docs.deleted store.size pri.store.size yellow open fulltext 5 1 2 0 10.9kb 10.9kb yellow open fulltext\_opt 1 1 2 0 159b 159b

That’s it, our documents where copied from one index to the other. Now you can remove the index created by mongo-connector if you want. The last step is to try out our new index and see if it really supports partial matching for our autocomplete function:

curl -H 'Content-Type: application/json' \  
 localhost:9200/fulltext\_opt/articles/\_search?pretty \  
 -d "{ \"query\": { \"match\": { \"title\": { \"query\": \"chi\", \"analyzer\": \"standard\" } } } }"{  
 "took" : 12,  
 "timed\_out" : **false**,  
 "\_shards" : {  
 "total" : 1,  
 "successful" : 1,  
 "failed" : 0  
 },  
 "hits" : {  
 "total" : 1,  
 "max\_score" : 0.3125,  
 "hits" : [ {  
 "\_index" : "fulltext\_opt",  
 "\_type" : "articles",  
 "\_id" : "579e0a35c6d02e54ad6fe556",  
 "\_score" : 0.3125,  
 "\_source" : {  
 "content" : "In the most recent example in a growing trend of big deals for smartphone-based games, a consortium of Chinese investors led by the game company Shanghai Giant Network Technology said in a statement on Saturday that it would pay $4.4 billion to Caesars Interactive Entertainment for Playtika, its social and mobile games unit. Caesars Interactive is controlled by the owners of Caesars Palace and other casinos in Las Vegas and elsewhere.",  
 "title" : "Chinese Group to Pay $4.4 Billion for Caesars Mobile Games"  
 }  
 } ]  
 }  
}

Voilà! Got our document back. Note that we defined in our query the specific analyser we wanted to use and set it to the standard one:

{   
 title: {   
 query: "chi",   
 analyzer: "standard"   
 }   
}

If we don’t do this, since we’re querying the index with our custom analyser, it would use the autocomplete analyser by default and query using the edge n-grams of the query text. This would lead to unwanted results, since we want to search for the text chi specifically, and not for c, or ch or chi. This is why we have to explicitly set the analyser to the standard one.

Handling new MongoDB inserts

So far we moved all our MongoDB collection contents to the fulltext\_opt index by using mongo-connector. The only problem is that, as you probably remember, mongo-connector copies to an index with the same database name from MongoDB. This means that if we keep mongo-connector running as we have it now, all the new documents inserted in the database will be indexed in the fulltext index in ES and not the optimized fulltext\_opt.

The way to solve this is by configuring a bit more the mongo-connector command. There are many configuration options that you can find [here](https://github.com/mongodb-labs/mongo-connector/wiki/Configuration%20Options). We’ll use two of them: namespaces.include (-nin command line) and namespaces.mapping (-g in command line). You can see how to configure mongo-connector through a json file, here I’ll just use the command line arguments way.

The -n option will tell mongo-connector which collections from MongoDB we want to index. The syntax for this is database\_name.collection\_name. In our case we want to index all the articles from the fulltext database. So we’ll pass a command line argument like this: -n fulltext.articles.

The -g option will tell mongo-connector into which index it should put all the documents taken from the collections defined with the -n option. So in our case we want to put all our articles in the fulltext\_opt index. We need to  
provide as well which type within ES we want to use, so the full argument would be: -g fulltext\_opt.articles, since we want our articles stored with the articles type in the index.

That’s it, now we can run the command like this:

$ mongo-connector -m 127.0.0.1:27017 -t 127.0.0.1:9200 -d elastic2\_doc\_manager -n fulltext.articles -g fulltext\_opt.articles

If you keep mongo-connector running, all new inserts will be indexed in ES as well. Go ahead and insert a new document in the articles collection and then send a query to the ES index, the document should be returned.

**Deliverable: - Handle Scaling in E-commerce Application.**

**How will you handle scaling, if required at any point of time?**

As the application grows, each piece of the application must scale along with the size of the data needs. In MongoDB there is **Vertical** **scaling** sand **Horizontal** **scaling**.

**Vertical scaling** refers to increasing the processing power of a single server or cluster. Both relational and non-relational database can scale up, but eventually there will be a limit in terms of maximum processing power and throughput. There are increased costs with scaling up to high-performing hardware, as costs do not scale linearly.

**Horizontal Scaling**, also known as scale-out, refers to bringing additional nodes to share the load. This is difficult with relational database due to the difficulty in spreading out related data across nodes. With non-relational database, this is made simpler since collections are self-contained and not coupled relationally. Scaling MongoDB horizontally is achieved through sharding and replica sets.

**In case of E-commerce Application, we would be using Horizontal Scaling to handle scaling.**

**Sharding is a technique used to scale database horizontally by distributing data across multiple servers or nodes. In MongoDB, sharding allows to partition the data based on a chosen “shard key” and distribute it across multiple shards, which are essentially separate MongoDB instances. This approach helps distribute the workload evenly among the shards, ensuring better performance, high availability, and fault tolerance.**

**Horizontal scaling**, refers to adding machines to share the data set and load. Horizontal scaling allows for near-limitless scaling to handle big data and intense workloads.

MongoDB sharding offers several benefits that which makes it an attractive option for scaling databases:

**Horizontal Scaling:** Unlike vertical scaling, which involves adding more resources to a single machine, horizontal scaling distributes data and workload across multiple machines. This allows for better performance and reduced risk of downtime.

**Fault Tolerance:** By distributing data across multiple shards, MongoDB sharding ensures that the data remains available even if on of the shards fails.

**Load Balancing:** Sharding helps distribute queries and write operations evenly among shards, preventing any single shard from becoming a bottleneck.

**Data Localization:** Sharding can be configured to store data close to the application servers that access it, reducing latency and improving performance.

**Components of a Sharded Cluster**

In MongoDB, sharded cluster consists of three main components:

1. Shards: These are individual MongoDB instances that store the actual data. Each shard contains a subset of the total data in the cluster, based on the shard key.
2. Config Servers: These servers store metadata about the cluster, such as the mapping of data to shards. Config servers ensure consistency across the cluster and are essential for the proper functioning of a sharded cluster.
3. Query Routers (mongos): Query routers act as intermediaries between clients and shards. They route queries and write operation, so the appropriate shards based on the shard key.

Choosing a Shard Key

The choice of a shard key is crucial, as it determines how data is distributed across the shards. A good shard key should:

1. Provide an even distribution of data and workload across shards.
2. Minimize the need for “chunk migrations”, which occur when data is moved between shards to maintain an even distribution.
3. Support the most common quey pattern in the application.

Some common choices for shard keys include:

* Hashed shard keys: Hashed shard keys provide an even distribution of data and workload but may not support range queries efficiently.
* Compound shard keys: Compound shard keys can be used to distribute data based on multiple fields, providing more flexibility in query routing.