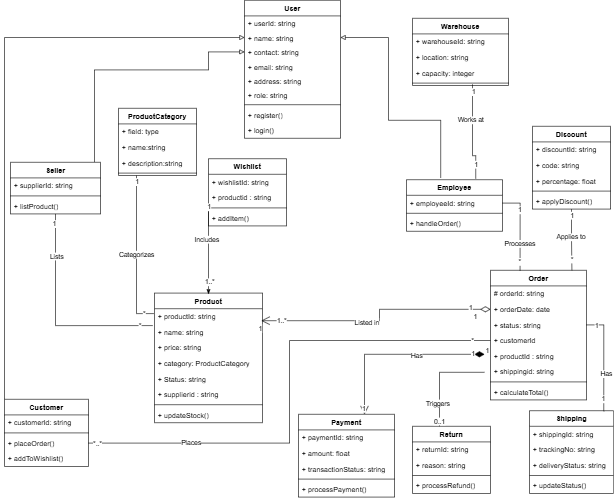
**Task One:**



**Rationale**

The embeddings of complex relationships in the UML diagram above serve three key purposes: **domain accuracy**, **system flexibility**, and **data integrity**. The Customer, Employee, and Seller entities' inheritance dependency on the User class allows for polymorphic user management, reducing some attributes' duplication (Baniassad, 2018). The composition of Order and Product ensures coupling which ensures that items in an Order cannot exist independently, which is a critical business rule in transactional systems (Peri, et al., 2018).

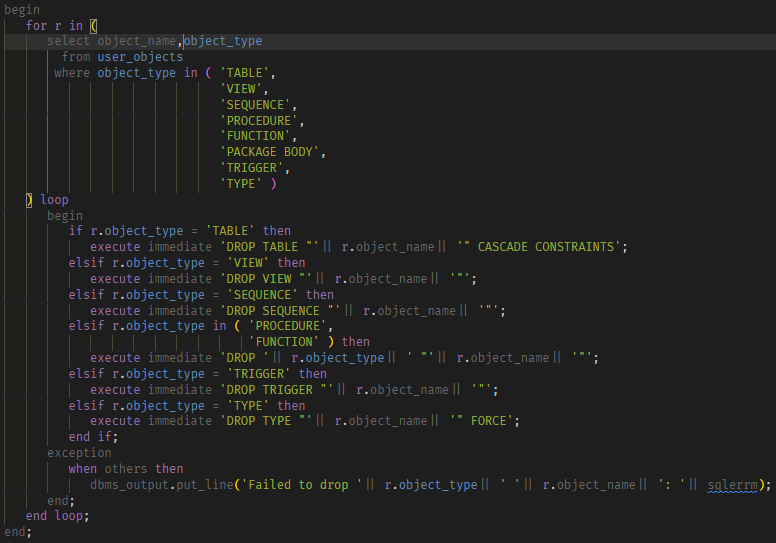
The ternary relationship between Employee, Order, and Warehouse captures operational metadata that tracks employees involved in order processing. The use of composition on the Payments object enforces the tight link with orders which ensures data integrity.

Many-to-many associations allow for complex promotional strategies while still maintaining auditability through composition on the Payment entity. This model avoids an overuse of inheritance beyond the User hierarchy instead, it employs interface-like patterns. This avoidance favours a composite approach which allows for the flexibility required in NoSQL implementation (MongoDB) while maintaining relational rigor.

**Task Two:**

The following screenshots show a sample of the types, triggers, table creation and stored procedure statements used in the database. There are more, lengthier, and more complex implementations in the SQL file that couples this document.

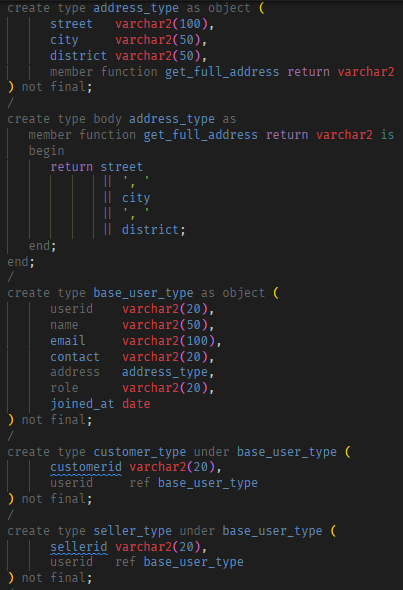
**Dropping all items**

****

**Output**

****

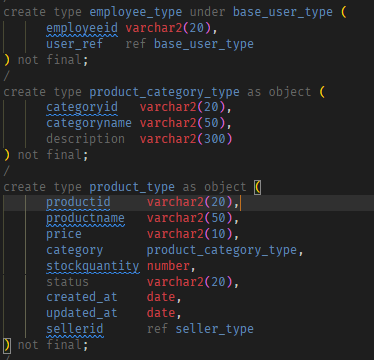
**Type Creations**

****

**Output**

**A screenshot of a computer program

AI-generated content may be incorrect.**

****

**Output**

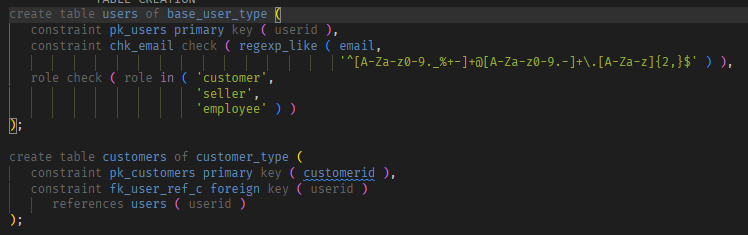
**A screen shot of a computer

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Table Creation statements**

****

**Output**

**A black screen with white text

AI-generated content may be incorrect.**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Output**

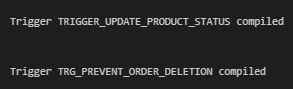
****

**Triggers**

**A computer screen shot of a program

AI-generated content may be incorrect.**

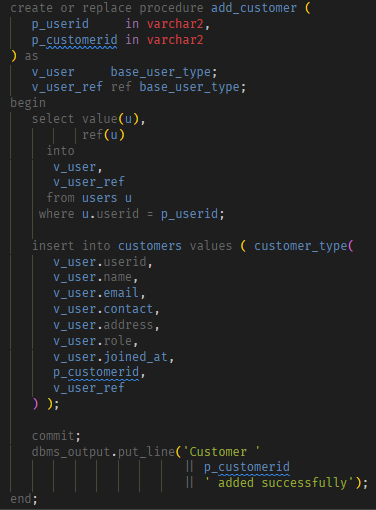
**Output**

****

**A screen shot of a computer error

AI-generated content may be incorrect.**

**Stored Procedures**

****

**Implementation and output**

****

**A screenshot of a computer program

AI-generated content may be incorrect.**

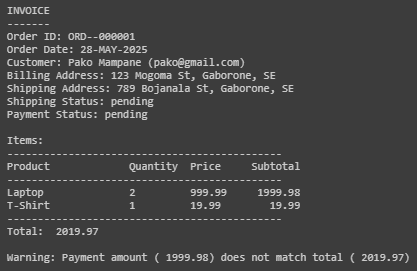
**A black background with white text

AI-generated content may be incorrect.**

**Functions**

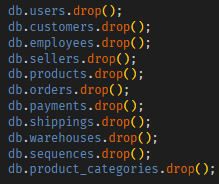
The SQL script contains a function called generate\_invoice that is found between lines 688 and 865 that generates an invoice for any given order ID. The output of the function is as follows:

* Function call: 
* Result:



**Task Three:**

**Dropping collections**

****

**Creating Collections**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**A screen shot of a computer

AI-generated content may be incorrect.**

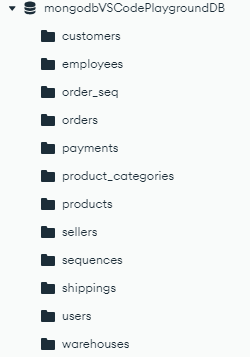
**A computer screen shot of code

AI-generated content may be incorrect.**

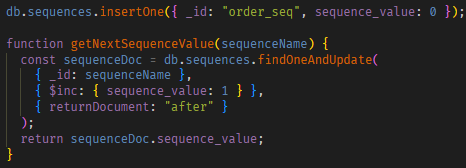
**A screen shot of a computer program

AI-generated content may be incorrect.**

**Output of collection Creation**

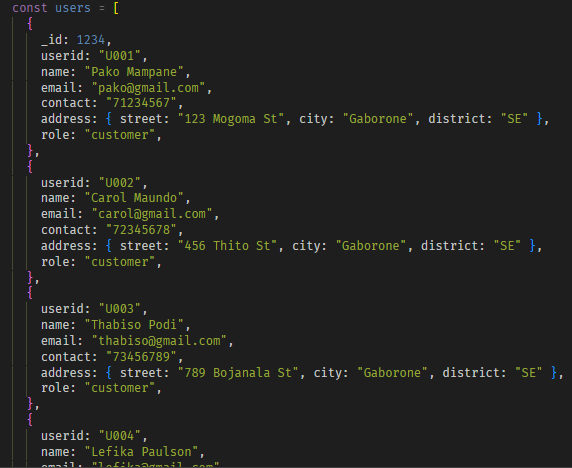
****

**Sequence Creation**

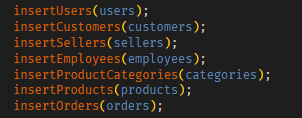
****

**Insertion Statements**

Arrays

****

Function calls to insert the data.

****

**Triggers**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Implementation of Stored Procedures**

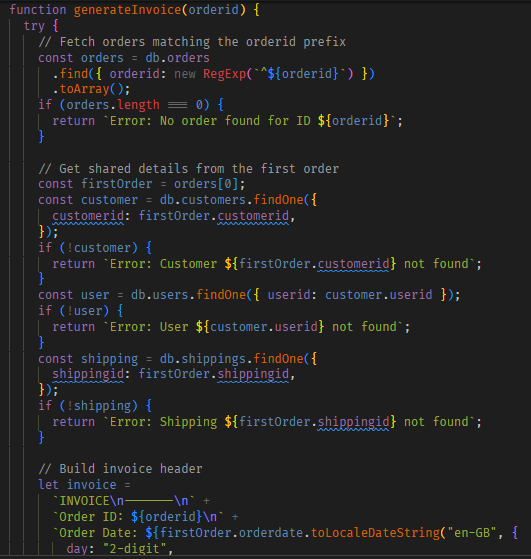
**A computer screen with colorful text

AI-generated content may be incorrect.**

**A screen shot of a computer program

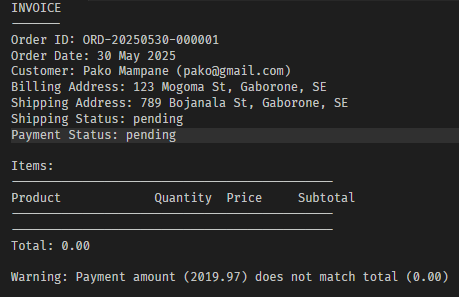
AI-generated content may be incorrect.**

**Function to generate an invoice**

****

The rest of the function between the lines 406 and 487.

**Function Output**



**Task Four**

Converting the object-relational database to a MongoDB NoSQL document store leveraged MongoDB’s document related features to improve robustness and reduce complexity. MongoDB’s document model eliminated the need for many tables by allowing embeddings of relational data, for example, the address\_type was merged with users. This helped to eliminate the need for joins thus simplifying the schema and reduced the look up latency (MongoDB, n.d.). This denormalization improves read efficiency while leading to data redundancy, however, MongoDB mitigates this through atomic updates on subdocuments

The need for multiple rows in the orders table was removed thanks to MongoDB’s allowance for consolidation of many rows into an array. For example, the productid column contains the values [“P001”, “P002”] in a single document instead of two. This reduced the overall document count and simplified querying.

MongoDB’s JSON schema allowed for the enforcement of constraints similar to those in SQL. For example, an enum was set on the status category of products which mimicked the CHECK and a regular expression check on the email field on the users collection. This optional and less rigid nature of schema enforcement offers flexibility to evolving data models without sacrificing data quality (MongoDB, n.d.). The change streams functionality replicated triggers present in the object relational database like the watchProductStatus which throws an error if an invalid status value is entered.

Aggregation pipelines were used in place of functions like generate\_invoice in SQL. These aggregation pipelines eliminated the need to join data with REF types which enhances flexibility when executing queries.

This migration to a NoSQL MongoDB introduces simplification and agility to schema design with careful considerations in data redundancy and update complexity. This document model would be more robust in read-heavy operations requiring flexible data structures.

**Tasks Five and Six**

*Complete the following tables:*

|  |  |
| --- | --- |
| **Query a: A join of three or more tables – you should consider various types of join in this query (e.g. inner join, left/right/full outer joins, etc.) and the query must include a restriction on the rows selected**  This query fetches order details for customers located in the SE (South East) district. It achieves this by joining the customers, payments, and shippings table using object references. The results are ordered from the most recent ones. | |
| **SQL code** | **MongoDB code** |
| select o.orderid,         o.orderdate,         o.status as order\_status,         c.name as customer\_name,         c.address.city as customer\_city,         p.transactionstatus as payment\_status,         s.trackingno,         s.deliverystatus as shipping\_status,         s.address.city as shipping\_city    from orders o   inner join customers c  on o.customerid = ref(c)    left join payments p  on o.payment.paymentid = p.paymentid    left join shippings s  on o.shipping = ref(s)   where c.address.district = 'SE'   order by o.orderdate desc; | db.orders.aggregate([    {      $lookup: {        from: "customers",        localField: "customerid",        foreignField: "customerid",        as: "customer",      },    },    { $unwind: "$customer" },    {      $lookup: {        from: "users",        localField: "customer.userid",        foreignField: "userid",        as: "user",      },    },    { $unwind: "$user" },    {      $match: { "user.address.district": "SE" },    },    {      $lookup: {        from: "payments",        localField: "payment.paymentid",        foreignField: "paymentid",        as: "payment\_doc",      },    },    {      $unwind: {        path: "$payment\_doc",        preserveNullAndEmptyArrays: true,      },    },    {      $lookup: {        from: "shippings",        localField: "shippingid",        foreignField: "shippingid",        as: "shipping",      },    },    {      $unwind: {        path: "$shipping",        preserveNullAndEmptyArrays: true,      },    },    {      $project: {        \_id: 0,        orderid: 1,        orderdate: 1,        order\_status: "$status",        customer\_name: "$user.name",        customer\_city: "$user.address.city",        payment\_status: "$payment\_doc.transactionstatus",        trackingno: "$shipping.trackingno",        shipping\_status: "$shipping.deliverystatus",        shipping\_city: "$shipping.address.city",      },    },    { $sort: { orderdate: -1 } },  ]); |
| **Screenshots** | |
|  | |
|  | |

|  |  |
| --- | --- |
| **Query b: A query which uses one (or more) of the UNION, DIFFERENCE or INTERSECT operators.**  This query fetches a list of active customers and sellers in Gaborone and Francistown. The union operator is used to merge similar records from the customers and sellers tables then assign a user type label to each. | |
| **SQL code** | **MongoDB code** |
| select c.customerid as id,         c.name,         c.email,         c.contact,         'Customer' as user\_type,         c.address.city as city    from customers c   where c.address.city in ( 'Gaborone',                             'Francistown' )     and c.role = 'customer'  union  select s.sellerid as id,         s.name,         s.email,         s.contact,         'Seller' as user\_type,         s.address.city as city    from sellers s   where s.address.city in ( 'Gaborone',                             'Francistown' )     and s.role = 'seller'   order by city,            name; | const customers\_agg = db.customers.aggregate([  {  $lookup: {  from: "users",  localField: "userid",  foreignField: "userid",  as: "user"  }  },  { $unwind: "$user" },  {  $match: {  "user.address.city": { $in: ["Gaborone", "Francistown"] },  "user.role": "customer"  }  },  {  $project: {  id: "$customerid",  name: "$user.name",  email: "$user.email",  contact: "$user.contact",  user\_type: { $literal: "Customer" },  city: "$user.address.city"  }  }  ]).toArray();  const sellers\_agg = db.sellers.aggregate([  {  $lookup: {  from: "users",  localField: "userid",  foreignField: "userid",  as: "user"  }  },  { $unwind: "$user" },  {  $match: {  "user.address.city": { $in: ["Gaborone", "Francistown"] },  "user.role": "seller"  }  },  {  $project: {  id: "$sellerid",  name: "$user.name",  email: "$user.email",  contact: "$user.contact",  user\_type: { $literal: "Seller" },  city: "$user.address.city"  }  }  ]).toArray();  const combined = [...customers\_agg, ...sellers\_agg];  combined.sort((a, b) => {  const cityA = a.city || "";  const cityB = b.city || "";  const nameA = a.name || "";  const nameB = b.name || "";  return cityA.localeCompare(cityB) || nameA.localeCompare(nameB);  });  combined.forEach(doc => printjson(doc)); |
| **Screenshots** | |
|  | |
|  | |

|  |  |
| --- | --- |
| **Query c: A query which requires use of either a nested table or subtypes**  This query fetches available products in the Electronics or Appliances categories. It accesses nested attributes from the inherited category object type. The results are then ordered by category and product names. | |
| **SQL code** | **MongoDB code** |
| select p.productid,         p.productname,         p.price,         p.status,         p.category.categoryname as category,         p.category.description as category\_description    from products p   where p.status = 'available'     and p.category.categoryname in ( 'Electronics',                                      'Appliances' )   order by p.category.categoryname,            p.productname; | db.products.aggregate([    {      $match: {        status: "available",        "category.categoryname": { $in: ["Electronics", "Appliances"] },      },    },    {      $project: {        productid: 1,        productname: 1,        price: 1,        status: 1,        category: "$category.categoryname",        category\_description: "$category.description",      },    },    {      $sort: {        category: 1,        productname: 1,      },    },  ]); |
| **Screenshots** | |
|  | |
|  | |

|  |  |
| --- | --- |
| **Query d: A query using temporal features (e.g., timestamps, intervals, etc.) of Oracle SQL**  This query provides an analysis of the average processing time for orders with a **pending payment** status in the last 6 months. It calculates the time taken between the order date and shipping update date of which results are grouped by product category, customer name, and order month. Only groups that contain atleast one order are included. | |
| **SQL code** | **MongoDB code** |
| select deref(o.product).category.categoryname as category,         deref(o.customerid).name as customer\_name,         extract(month from cast(o.orderdate as timestamp)) as order\_month,         count(o.orderid) as order\_count,         avg(extract(day from(cast(deref(o.shipping).updated\_at as timestamp) - cast(o.orderdate as timestamp))) + extract(hour         from(cast(deref(o.shipping).updated\_at as timestamp) - cast(o.orderdate as timestamp))) / 24.0 + extract(minute from(cast         (deref(o.shipping).updated\_at as timestamp) - cast(o.orderdate as timestamp))) /(24.0 \* 60.0) + extract(second from(cast         (deref(o.shipping).updated\_at as timestamp) - cast(o.orderdate as timestamp))) /(24.0 \* 60.0 \* 60.0)) as avg\_processing\_days    from orders o   where o.payment.transactionstatus = 'pending'     and o.orderdate >= sysdate - interval '6' month   group by deref(o.product).category.categoryname,            deref(o.customerid).name,            extract(month from cast(o.orderdate as timestamp))  having count(o.orderid) > 0   order by category,            customer\_name,            order\_month; | db.orders.aggregate([    {      $match: {        "payment.transactionstatus": "pending",        orderdate: {          $gte: new Date(new Date().setMonth(new Date().getMonth() - 6)),        },      },    },    {      $lookup: {        from: "customers",        localField: "customerid",        foreignField: "customerid",        as: "customer",      },    },    { $unwind: "$customer" },    {      $lookup: {        from: "users",        localField: "customer.userid",        foreignField: "userid",        as: "user",      },    },    { $unwind: "$user" },    {      $lookup: {        from: "shippings",        localField: "shippingid",        foreignField: "shippingid",        as: "shipping",      },    },    { $unwind: "$shipping" },    { $unwind: { path: "$productid", includeArrayIndex: "productIndex" } },    {      $lookup: {        from: "products",        localField: "productid",        foreignField: "productid",        as: "product",      },    },    { $unwind: "$product" },    {      $addFields: {        processing\_time: {          $divide: [            { $subtract: ["$shipping.updated\_at", "$orderdate"] },            1000 \* 60 \* 60 \* 24,          ],        },      },    },    // Group by category, customer name, and order month    {      $group: {        \_id: {          category: "$product.category.categoryname",          customer\_name: "$user.name",          order\_month: { $month: "$orderdate" },        },        order\_count: { $sum: 1 },        avg\_processing\_days: { $avg: "$processing\_time" },      },    },    // Filter groups with order\_count > 0    { $match: { order\_count: { $gt: 0 } } },    // Project final output    {      $project: {        category: "$\_id.category",        customer\_name: "$\_id.customer\_name",        order\_month: "$\_id.order\_month",        order\_count: 1,        avg\_processing\_days: 1,        \_id: 0,      },    },    // Sort by category, customer\_name, order\_month    {      $sort: {        category: 1,        customer\_name: 1,        order\_month: 1,      },    },  ]); |
| **Screenshots** | |
|  | |
|  | |

|  |  |
| --- | --- |
| **Query e: A query using OLAP (e.g., ROLLUP, CUBE, PARTITION) features of Oracle SQL**  This query analyses sales by product category and order status using the ROLLUP function. It displays the number of orders and total sales where the payment status is completed. The grouping and grouping\_id functions are used to identify subtotals and grand totals. | |
| **SQL code** | **MongoDB code** |
| select p.category.categoryname as category,         o.status as order\_status,         count(o.orderid) as order\_count,         sum(to\_number(p.price)) as total\_revenue,         grouping(p.category.categoryname) as category\_grouping,         grouping(o.status) as status\_grouping    from orders o    join products p  on p.productid = deref(o.product).productid   where o.payment.transactionstatus = 'completed'   group by rollup(p.category.categoryname,                   o.status)  having grouping\_id(p.category.categoryname,                     o.status) in ( 0,                                    3 )   order by p.category.categoryname,            o.status; | db.orders.aggregate([    { $match: { "payment.transactionstatus": "completed" } },    { $unwind: "$productid" },    {      $lookup: {        from: "products",        localField: "productid",        foreignField: "productid",        as: "product",      },    },    { $unwind: "$product" },    {      $lookup: {        from: "customers",        localField: "customerid",        foreignField: "customerid",        as: "customer",      },    },    { $unwind: "$customer" },    {      $group: {        \_id: {          category: "$product.category.categoryname",          order\_status: "$status",        },        order\_count: { $sum: 1 },        total\_sales: { $sum: "$product.price" },      },    },    {      $match: {        $or: [          { "\_id.category": { $ne: null }, "\_id.order\_status": { $ne: null } },          { "\_id.category": null, "\_id.order\_status": null },        ],      },    },    {      $project: {        category: "$\_id.category",        order\_status: "$\_id.order\_status",        order\_count: 1,        total\_sales: 1,        category\_grouping: {          $cond: [{ $eq: ["$\_id.category", null] }, 1, 0],        },        status\_grouping: {          $cond: [{ $eq: ["$\_id.order\_status", null] }, 1, 0],        },        \_id: 0,      },    },    {      $sort: {        category: 1,        order\_status: 1,      },    },  ]); |
| **Screenshots** | |
|  | |
|  | |

**References**

Baniassad, E., 2018. *Making the liskov substitution principle happy and sad.* s.l., Association for Computing Machinery.

MongoDB, n.d. *MongoDB - Database Manual - MongoDB Docs.* [Online]   
Available at: https://www.mongodb.com/docs/manual/data-modeling/

MongoDB, n.d. *Schema Validation - Database manual - MongoDB docs.* [Online]   
Available at: https://www.mongodb.com/docs/manual/core/schema-validation/

Peri, S., Singh, A. & Somani, A., 2018. Efficient Means of Achieving Composability Using Object Based Semantics in Transactional Memory Systems. *Indian Institute of Technology, Hyderabad,* pp. 157-174.