Alexander Holden (ath99), Franklin Wang (fmw13), Zachary Ansell (zra11)
Sales Orders Database Team 2

The “Sales Orders Database” keeps information about US businesses’ sales orders. The information the database retains includes customers, their orders, the order details, products available for sale, suppliers of those products, and businesses who sell products.

The Sales Orders Database

Table of Contents

Detailed Instructions on How to Reproduce the Project

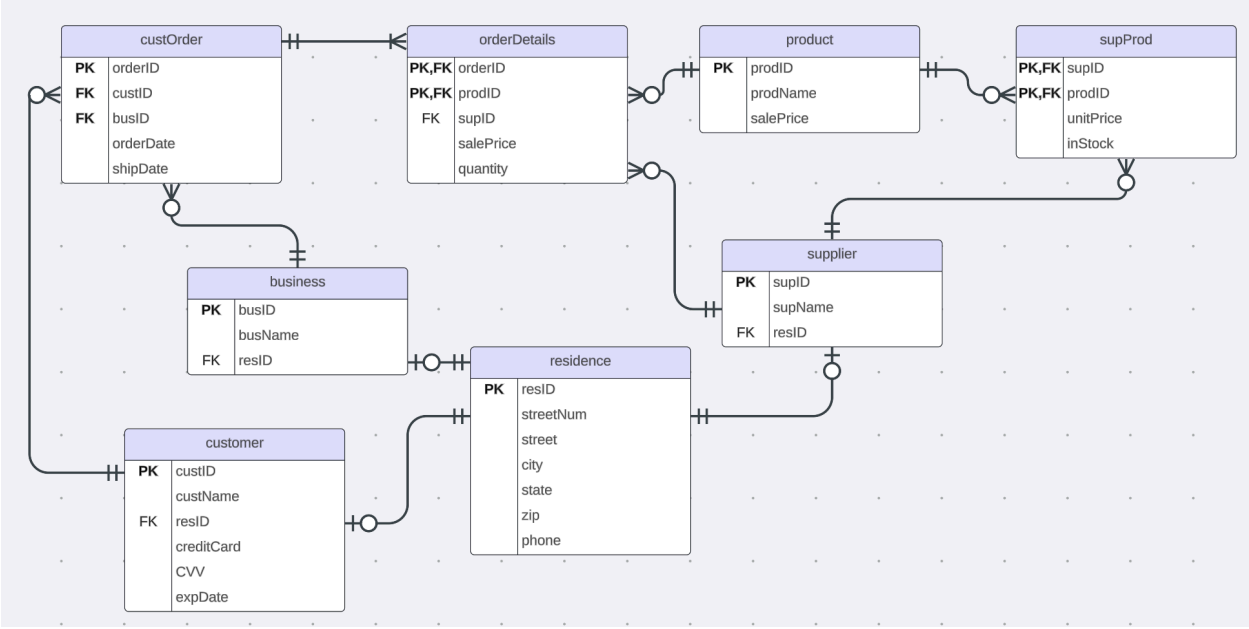
Database backup file: backup.bak
Zip file name with java code and SQL files: deliverables.zip
TA: Emil
Demo Time: December 6th, 4:00pm
Demo Leader: Franklin Wang

Accessing SQL Server Virtual Environment
Team member: Franklin Wang fmw13
SA password: T1eUtHba5cotjx
SQL Server Name: cxp-sql-02
Database Name: salesOrders
Database username: dbuser
Database password: csds341143sdsc

Other information:
The database name in backup.bak is salesDB
This backup does not have the stored procedures for use case 5 and 6. Please open StoredProc_for_5_and_6.sql and run the queries separately to create the stored procedures in the backup and grant permission.

Description of the Database

ERD



Explanation of Tables

Schema
customer(<u>custID</u>, custName, resID, creditCard, CVV, expDate)
business(<u>busID</u>, busName, resID)
supplier(<u>supID</u>, supName, resID)
product(<u>prodID</u>, prodName, salePrice)
supProd(<u>supID</u>, <u>prodID</u>, unitPrice, inStock)
custOrder(<u>orderID</u>, custID, busID, orderDate, shipDate)
orderDetails(<u>orderID</u>, <u>prodID</u>, supID, salePrice, quantity)
residence(<u>resID</u>, streetNum, street, city, state, zip, phone)

The customers table includes custID, the unique identifier of each customer starting at 1000 and increments by 1 each time. The customer name as custName, their residence, and credit card information split up into number, cvv, and expiration date. No field may be null.

The business table includes the unique business identification busID starting at 5000 and incrementing by 1 each time. It includes the business name as busName, and their residence. No field can be null

The supplier table includes the supplierID which starts at 200 and increments by 1. It includes their name as supName, and their residence. No field can be null.

The product table includes the product identification starting at 200000, the name, and the sale price of each product. No field can be null.

The supProd table tracks what products a supplier provides with the prodID and supID. It also includes the unitPrice, the price the suppliers supply the product at, and the quantity in stock each supplier has. No field can be null.

The custOrder table includes orderID, the unique identifier of each order starting at 10000 and increments by 1 each time. The customer who ordered as custID, the business who sold it as busID, the order date, and ship date. shipDate may be null, as this means an order has been placed but yet to be shipped, but it should be greater than or equal to the orderDate.

The orderDetails table includes the prodID for each orderID, the sale price of each product sold, the quantity of that product sold, and the supplier of that product. No two suppliers can supply the same product in the same order. No field may be null. A trigger is created so that inStock in the supProd table automatically decreases accordingly to the quantity sold. If the orderDetail is updated, the supProd table should handle changes accordingly, and if the supplier does not have any in stock, the transaction will rollback.

The residence table includes the unique resID, the street number as streetNum, street name as street, city, state, and zip, a phone number. This table is a collection of all residences for customers, businesses, and suppliers. resID begins at 70000 and increases by one. Phone number may be null, other fields cannot. An index is created on the zip code attribute because shipping addresses are referenced constantly and so it is preferable for addresses to have fast access. The zip code attribute is preferred as it provides geographic information on the state and city. Two or more customers, businesses, or suppliers may share a single residence.

Relationships Between Tables and Business Rules

Business Rules

- An order must have one and only customer and one and only one business that handles the order.
- Customers may share the same credit card information.
- Suppliers, businesses, and customers must have one and only residence.
- Suppliers can choose to supply as many or zero products because a supplier can choose to discontinue to supply their product.
- An orderDetail must have at least one product, and it can have many products. For example, a customer can buy carrots (one product) at a grocery store (business), or buy carrots and lettuce (two products).

- Businesses need to be able to know which supplier is supplying a product and suppliers may supply similar products at different prices. For example, carrots with prodID 111 and carrots with prodID 222 can have different unitPrices, even though they have the same product name.
- A business does not need to handle an order to be a business because they may be a start-up or newly registered.
- The unit price of a product from a supplier to a business is the price the supplier supplies the product to the business. It is not necessarily the same as the sale price that the business sells the product to a customer.
- Customers are always addressed by full name because the name is used for shipping, meaning it is unnecessary to split up the custName entity.
- The business will ship the order to the customer's residence in the residence table.
- The ship date cannot precede the order date, but they can be on the same date if a customer bought something in person.
- Credit card information is stored with the customer as the credit card lenders resolve security concerns.
- An order can be provided by multiple suppliers, but they cannot supply the same product. For example, given an order with products: apples, oranges, and bananas, and two suppliers A, B. A can supply apples and oranges while B can supply bananas. They cannot both supply apples, or both supply oranges in the same order. The reason for allowing multiple suppliers is because not all suppliers can supply all the products in the order. We disallow having different suppliers supplying the same product because it is not typical for a customer to order the same product from different suppliers.
- A business may share the same residence as a customer or supplier, or any combination of that, as the entity may sell items, supply items, or purchase items and be in all three tables as result. Their identification in each table must be unique in those tables however.

Table Relationships
supplier to supProd 1-(0,M): A supplier can supply many products, or none at all if they are a newly registered supplier.

product to supProd 1-(0,M): A product can be supplied by many suppliers, or none at all if the product is discontinued.

custOrder to customer (0,M)-1: A customer can place many orders, or none at all. An order can be placed by one customer.

custOrder to business (0,M)-1: A business can make many orders, or none at all. An order can be made by one business only.

orderDetails to order M-1: A order can be in many orderDetails as an order can have many products. An order must have at least one product. An order detail can contain 1 or many details about an order.

orderDetails to products (0,M)-1: The details of an order contains all the products in the single order described. A single product can be ordered 0 or many times. Each detail can list a single product at a time.

orderDetails to supplier (0,M)-1: A supplier can supply 0 or many products to an order in the order detail. The product in the order detail must be supplied by one and only one supplier.

residence to supplier 1-(0,1): A supplier has exactly one residence. A residence can belong to a supplier or not (it may belong to a customer or business).

residence to customer 1-(0,1): A customer has exactly one residence. A residence can belong to a customer or not (it may belong to a supplier or business).

residence to business 1-(0,1): A business has exactly one residence. A residence can belong to a business or not (it may belong to a customer or supplier).

Use Cases

Use Case 1: Update Customer Address

If a customer were to change their address, we would need to update the information inside their associated tuple in the residence table according to the custID.

Stored Procedures:

```
-- to create updateCustomeResidence stored procedure
create or alter procedure updateCustomerResidence
	@custID as int,
	@streetNum as varchar(20),
	@street as varchar(20),
	@city as varchar(20),
	@state as varchar(20),
	@zip as varchar(5),
	@phone as varchar(10)
as
begin
	begin tran tranUpdCustRes
		update residence set streetNum = @streetNum, street = @street, city = @city,
state=@state, zip=@zip, phone=@phone
		where (resID = (select resID from customer where custID = @custID))
	commit tran tranUpdCustRes;
end
```

```
--To grant permission for stored procedure
GRANT execute on object::updateCustomerResidence to dbuser

--To execute stored procedure
exec updateCustomerResidence 1000, '99', 'Whitney', 'Short Hills', 'NJ', '07078', '9735684482';
```

Use Case 2: Inserting a New Order

A customer places an order for an item and it needs to be populated into the custOrders table, and the items need to be put into the orderDetails table. To do this, we first run a stored procedure that inserts into the custOrder table, and returns the orderID. Then, using that orderID, we run a stored procedure to insert into the orderDetails table. Additionally, a trigger than activates which updates the stock of the ordered products in the supProd table.

Stored Procedures:

```
--To insert into custOrder table
create or alter procedure insertCustOrder
	@custID as int,
	@busID as int,
	@orderDate as date,
	@shipDate as date,
	@ID int OUTPUT
as
begin
	begin tran insCustOrder
		insert into custOrder (custID, busID, orderDate, shipDate)
		values (@custID, @busID, @orderDate, @shipDate);
		set @ID = scope_identity();
	commit tran insCustOrder
end;

--To insert into orderDetails table
create or alter procedure insertOrderDetails
	@orderID int,
	@prodID int,
	@supID int,
	@salePrice int,
	@quantity int
as
begin
```

```
        begin tran tranInsOrdDetails
                insert into orderDetails (orderID, prodID, supID, salePrice, quantity)
                values (@orderID, @prodID, @supID, @salePrice, @quantity);
        commit tran tranInsOrdDetails
end;

--To grant permissions for stored procedures
grant execute on object::insertCustOrder to dbuser;
grant execute on object::insertOrderDetails to dbuser;

--To execute procedures
begin
declare @orderID int;
exec insertCustOrder
1000,
5000,
'11-11-2024',
'11-12-2024',
@orderID output;

exec insertOrderDetails
@orderID  = @orderID,
@prodID = 200000,
@supID = 200,
@salePrice = 1800,
@quantity = 3;
end;
```

Use Case 3: Inserting a New Business with a New Residence

A newly registered business with a new residence needs to be included in the database. To do this, a new residence needs to be inserted first and the resID needs to be returned. Then the business needs to be inserted into the business table along with the returned resID. This use case then touches two tables.

Stored Procedures:

```
--To create the stored procedure for new residence
CREATE or ALTER procedure insertNewResidence
        @streetNum as int, -- input parameter
        @street as varchar(20), -- input parameter
        @city as varchar(20), -- input parameter
        @state as varchar(20), -- input parameter
        @zip as varchar(5), -- input parameter
        @phone as varchar(20), -- input parameter
```

```
	@id int OUTPUT
as
	begin
		begin tran insertResidence
		insert into residence (streetNum, street, city, state, zip, phone)
		values (@streetNum, @street, @city, @state, @zip, @phone)
		set @ID = scope_identity()
		commit tran insertResidence
	end;

--To execute new residence procedure in SQL (assuming permissions are granted)
begin
	declare @iid int;
	EXEC insertNewResidence
		3450,
		'178th Ave',
		'Portland',
		'OR',
		'97229',
		'5039115987',
		@iid OUTPUT;
	select @iid;
end;

--To create the stored procedure for new business
CREATE or ALTER procedure insertNewBusiness
	@name as varchar(20),
	@resID as int,
	@id int OUTPUT
as
	begin
		begin tran insertBusiness
			insert into business (busName, resID)
			values (@name, @resID)
			set @ID = scope_identity()
		commit tran insertBusiness
	end;

--To execute new business procedure in SQL (assuming permissions are granted)
begin
	declare @iid int;
	EXEC insertNewBusiness
		'The Spot',
		70003, --should be the same as what was printed previously
```

```
            @iid OUTPUT;
        select @iid;
end;
```

After creating dbuser in SQL, permission to execute must be granted:

```
GRANT EXECUTE ON OBJECT::insertNewResidence TO dbuser
GRANT EXECUTE ON OBJECT::insertNewBusiness TO dbuser
```

Transaction control is not necessary for this use case as it is two separate insert cases that are independent of each other. Transaction control is also implemented in the java code.
This use case could be combined into a single stored procedure, but I think that would just make it unnecessarily complex. It could also be combined into one menu option in the java code presented in the next section. This is done by saving the newly generated ID in a variable and passing it along to the parameters of inserting a new business. This was not implemented because it was not necessary for the use case to function. The java code and screenshots are visible in the individual assignment.

Use Case 4: Finding Local Suppliers

A business wants to search for which supplier to use that might be close to where their customers are. To identify suppliers in the same area as a given customer, we can retrieve data from both the suppliers and customers tables based on a matching zip code. This requires a select statement with a join between the suppliers, customers, and residences tables, filtered by the zip code attribute, which is indexed for fast access.

Stored Procedure:

```
--To create the stored procedure for a local supplier
CREATE or ALTER procedure findLocalSupplier
        @custID as int -- input parameter
as
        begin
                begin tran findSup
                        SELECT supID
                        FROM customer join residence on customer.resID = residence.resID
                        join supplier on supplier.resID = residence.resID
                        WHERE customer.custID = @custID
                commit tran findSup
        end;

--To execute findLocalSupplier procedure in SQL (assuming permissions are granted)
begin
        declare @iid int;
        EXEC findLocalSupplier
```

```
                @custID = 1000
end;
```

Grant executable connection:

```
        GRANT EXECUTE ON OBJECT::findLocalSupplier TO dbuser
```

Transaction control is not necessary for this use case as it is a select statement that does not alter the database. The java code and screenshots are in the individual assignment.

Use Case 5: Canceling an Order

A customer may want to cancel an order. To do this, the orderID needs to be entered for the order to be cancelled. The orderID should also be saved so that each entry in the orderDetails table also gets deleted. If the order was shipped already, cancelling the order

Stored Procedure:

```
CREATE or ALTER procedure cancelOrder
        @orderID as int
as
begin
        begin tran cancelOrder
                delete from orderDetails
                where orderID = @orderID;

                delete FROM custOrder
                WHERE orderID = @orderID;
        commit tran cancelOrder
end;

--To execute procedure
EXEC cancelOrder @orderID = 10002
```

Granting Permission:

```
        GRANT EXECUTE ON OBJECT::cancelOrder TO dbuser
```

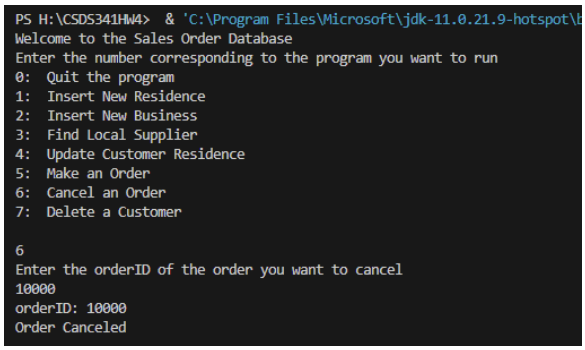
Transaction control is implemented so that the change is only committed if both the entries in the orderDetails table AND the entry in custOrders is removed successfully. The trigger implemented for auto-decreasing the inStock attribute will no longer be accurate, and so another trigger should be implemented here to adjust for that. The solution for this logical error is noted, but will not be implemented.

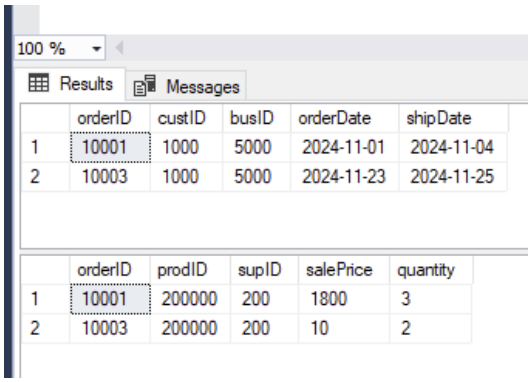
Java Code

```
   public static void cancelOrder(){
```

```
        //supplier same zip as custID
        int orderID;

        //inputs
        System.out.println("Enter the orderID of the order you want to cancel");
        orderID = myObj.nextInt();

        System.out.println(
            "orderID: " + orderID
        );

        String cancelOrder = "{call dbo.cancelOrder(?)}";

        try (Connection connection = DriverManager.getConnection(connectionUrl);
            CallableStatement prepsStoredProc = connection.prepareCall(cancelOrder);) {
                prepsStoredProc.setInt(1, orderID);

                prepsStoredProc.execute();
                System.out.println("Order Canceled");
            }
        catch (SQLException e) {
            e.printStackTrace();
        }
        System.out.println();

    }
```

Java GUI Proof and Working:





Use Case 6: Finding the Total Sales Made to a Customer

A business may want to know the total amount of sales they made to a customer. To do this, we need to join the customer table, business table, the custOrder table, and the orderDetails table

based on the custID and busID, and the orderID. Then we need to sum the salePrice multiplied by the quantity of the entries in orderDetails.

Stored Procedure:

```
CREATE or ALTER procedure findTotalSales
        @custID as int,
        @busID as int
as
begin
        begin tran findTotalSales
                select custOrder.custID, sum(salePrice*quantity) as 'totalSales'
                from custOrder join customer on customer.custID = custORder.custID
                join business on custOrder.busID = business.busID
                join orderDetails on custOrder.orderID = orderDetails.orderID
                where custOrder.custID = @custID and custOrder.busID = @busID
                group by custOrder.custID
        commit tran findTotalSales
end;

EXEC findTotalSales @custID = 1000, @busID = 5000

GRANT EXECUTE ON OBJECT::findTotalSales TO dbuser
```

Java Code:

```
   public static void findTotalSales(){

       int custID;
       int busID;

       //inputs
       System.out.println("Enter the custID of the customer");
       custID = myObj.nextInt();
       System.out.println("Enter the busID of the business");
       busID = myObj.nextInt();

       System.out.println(
          "custID: " + custID + "\n" +
          "busID: " + busID
       );

       String findTotalSales = "{call dbo.findTotalSales(?,?)}";

       ResultSet resultSet = null;
```

```
        try (Connection connection = DriverManager.getConnection(connectionUrl);
            CallableStatement prepsStoredProc = connection.prepareCall(findTotalSales);) {
                prepsStoredProc.setInt(1, custID);
                prepsStoredProc.setInt(2, busID);

                resultSet = prepsStoredProc.executeQuery();

                System.out.println("CustID      TotalSales");
                boolean b;
                while (b = resultSet.next()) {
                    System.out.println(resultSet.getString(1) + "        " +
                    resultSet.getString(2));
                }
                if(!b){System.out.println("End");}

            }
        catch (SQLException e) {
            e.printStackTrace();
        }
        System.out.println();

    }
```

Proof:

```
Welcome to the Sales Order Database
Enter the number corresponding to the program you want to run
0:  Quit the program
1:  Insert New Residence
2:  Insert New Business
3:  Find Local Supplier
4:  Update Customer Residence
5:  Make an Order
6:  Cancel an Order
7:  Find the Total Sales Made to a Customer by a Business

7
Enter the custID of the customer
1000
Enter the busID of the business
5000
custID: 1000
busID: 5000
CustID      TotalSales
1000        5420
End
```

```
Select * from customer
select * from business
Select * from custOrder
select * from orderDetails
```

100 %

Results | Messages

	custID	totalSales
1	1000	5420

	custID	custName	resID	creditCard	CVV	expDate
1	1000	Alex Holden	70002	98377189028	4472	2027-01-11

	busID	busName	resID
1	5000	Franklin Inc.	70000
2	5002	Walmart	70005
3	5003	Fem LLC	70007
4	5004	Fem LLC	70007
5	5070	The Spot	70003
6	5071	ggg	70009

	orderID	custID	busID	orderDate	shipDate
1	10001	1000	5000	2024-11-01	2024-11-04
2	10003	1000	5000	2024-11-23	2024-11-25

	orderID	prodID	supID	salePrice	quantity
1	10001	200000	200	1800	3
2	10003	200000	200	10	2

This use case can also be modified so that it selects all the customers and the total sales for a single business by removing the where clause and the custID input. The group by clause is there for this reason. The transaction is implemented in SQL, though it is a select query.

Theoretical Use Cases:

1. A business wants to search for which supplier to use that might be close to where their customers are. To identify suppliers in the same city as a given customer, we can retrieve data from both the suppliers and customers tables based on a matching city value. This requires a select statement with a join between the suppliers, customers, and residences tables, filtered by the city attribute.

2. A shipper has lots of deliveries in one city, and so they need a list of all the addresses of the delivery for the delivery date. To do this, a select statement that joins the orders, customers, and residences tables, filtered by order date and by city, and returns the delivery addresses.

3. A data analyst wants to analyze certain purchasing patterns. To generate a monthly revenue report for each product name by region, we can retrieve data from the orders, orderDetails, products, residence, and customers tables. This requires a select statement that joins orders with orderDetails to access product details, joins products to access product name, joins customers and residences to access the region information, and groups by product name and region, and filters by orderDate. Finally, we can calculate the sum of revenue per group by subtracting the net salePrice and net unitPrice to complete the analysis.

4. A business wants to recommend similar products to customers who purchase a given product. To list all other products ordered alongside a specific product within the same order, we can retrieve data from the orders, orderDetails, and products tables. This requires a select statement that joins orderID to the orders table, filters for orders containing the specified productID, excludes the specified product itself, and retrieves distinct productIDs for all other products in those orders.

5. A business may have to do extra paperwork when making orders involving multiple suppliers, so it wants to detect orders that are problematic in this regard. To identify orders containing items from multiple suppliers, we can retrieve data from the orders, orderDetails, products, and suppliers tables. This requires a select statement that: joins orders with orderDetails to access each item in an order, joins products and suppliers to determine the supplier for each item, groups by orderID, and checks if there is more than one unique supplier in each order.

6. A supplier can have some production delays, so businesses may want to notify customers with pending orders that their order will be delayed. To identify customers with such orders, we can retrieve data from the orders, orderDetails, suppliers, and customers tables. This requires a select statement that joins orders with orderDetails and suppliers to find orders from that supplier where the current date is between the order date and ship date, or the ship date is null, and joins that with customers to find which customers ordered them.

7. A customer wants to find which businesses are faster at getting orders out and shipped than others. To identify the faster businesses, we can retrieve data from the orders, orderDetails, and business tables. This requires a select statement that joins orders and orderDetails by orderID, groups the results by business, and gives an average difference between orderDate and shipDate.

8. A business wants to serve targeted ads to customers for luxury goods, so it needs to identify customers who make orders of high-value products. To identify such customers, we can retrieve data from the customers, orders, and orderDetails tables. This requires a statement that joins orders and orderDetails to match them, joined with customers to group by the customer who made the order. We can then filter each group of orders based on the number of purchases above a certain price threshold, and remove customers who do not meet said threshold.

Functional Dependencies, Physical Database Design, and Normalization

Functional Dependencies
orderID → custID, busID, orderDate, shipDate
orderID, prodID → supID, salePrice, quantity
prodID → prodName, salePrice
supID, prodID → unitPrice, inStock
supID → supName, resID
resID → streetNum, street, city, state, zip, phone
custID → custName, resID, creditCardNum, CVV, expDate
busID → busName, resID

Physical Database Design
The physical database design defines tables, indexes, and storage structures, and selects data types, and keys considering the functional dependencies above. This is described in the logical schema in the Description of the Database.

Normalization
The tables in the Sales Order Database have atomic attributes to the extent of the business rules (custName may be split into first and last name, but our business has no use for it). All attributes are pure in that they have a single data type (1st normal form). The tables do not have partial dependencies because all attributes not part of the primary key are fully dependent on the primary key as shown in the functional dependencies section (2nd normal form). There exist no transitive dependencies. As a result, the database meets the requirements for 3rd normal form. We reduce redundancy (as in repeated attributes in different relations) by implementing a residence table which is intended to be a generalization of common attributes.

User Manual

Setting up the database:

Option 1) Make New Database

If making a new database, open Microsoft SQL and create a new database. Then run the DDL, and DML scripts to create and populate the database. For stored procedures, open the StoredProceduces.sql and run each procedure along. It will also be necessary to create a dbuser once you have the database. Click on security, and right click on users to create a new user. Then write dbuser for name and login name. The schema is dbo. Run the grant permission code in StoredProcedures.sql. Use cases 5 and 6 are not present in StoredProcedures.sql

Option 2) Restore Database from Backup

Follow these instructions to Restore a Database Backup Using SSMS. Then please open StoredProc_for_5_and_6.sql and run the queries separately to create the stored procedures in the backup and grant permission.

The Java Code
If necessary change the connection url with the appropriate login information, password, database name, and user. Run the java program.

Java Menu
The Java menu is self explanatory. Enter the number of the use case to enter as displayed on the prompt and enter 0 to quit the program.

Reflection

The Sales Orders Database is used to track businesses' sales orders, to increase traceability and transparency in the business transactions. The database is designed to help manage product information, order details, and store data for analytics of customer behaviors. This is done by tracking customer orders, and the products available.

As of report 2, each member has done the following: Alexander has completed the ER Diagram, functional dependencies, and business rules for the database. Franklin completed the physical database design, including the relational schema, and description of each table. Zachary described the use cases.

For this final report, Alexander completed the SQL implementation of the database, Franklin expanded on the topics of the second report, including normalization and functional dependencies as well as the schema and business rules. Zachary implemented the JDBC. The individual use cases were completed individually and tested together. The team distributed the work fairly and worked together to complete the final project.