File Organization and Database Management Midterm Exam

Saldaña, Javier :

Instructions

This is the midterm exam for MSDS 7330, File Organization and Database Management. This document contains questions for the exam. For your answers, create a Word document that clearly identifies every question number and your answer to that question including your intermediate solutions. Name the Word file containing your answers ‘yourLastName\_MSDS7330-20181021.doc’. Be sure to have your name on the first page of the Word file. Turn in the Word file to 2DS.

# Question (20 Points)

The 2018 college Football Bowl Series (FBS) contains 11 different conferences and a total of 130 college teams. Design a database schema to hold all the relevant information pertaining to the FBS including, but not limited to, teams, conferences, schedules, bowl games, and any rivalry trophies that teams may play for on a regular basis. Your design should be able to handle all games, conferences, etc from the beginning of college football until the end of time.

In addition to capturing all the information into an ER model, explain how your design can address the following requirements (you can provide sample SQL queries or explain query logic in plain English)

* + There should be a mechanism to capture all the games played by a given team



* + Capture all the games at a given venue (including venues that host Bowl Games)



* + Capture the players for each team with their captains

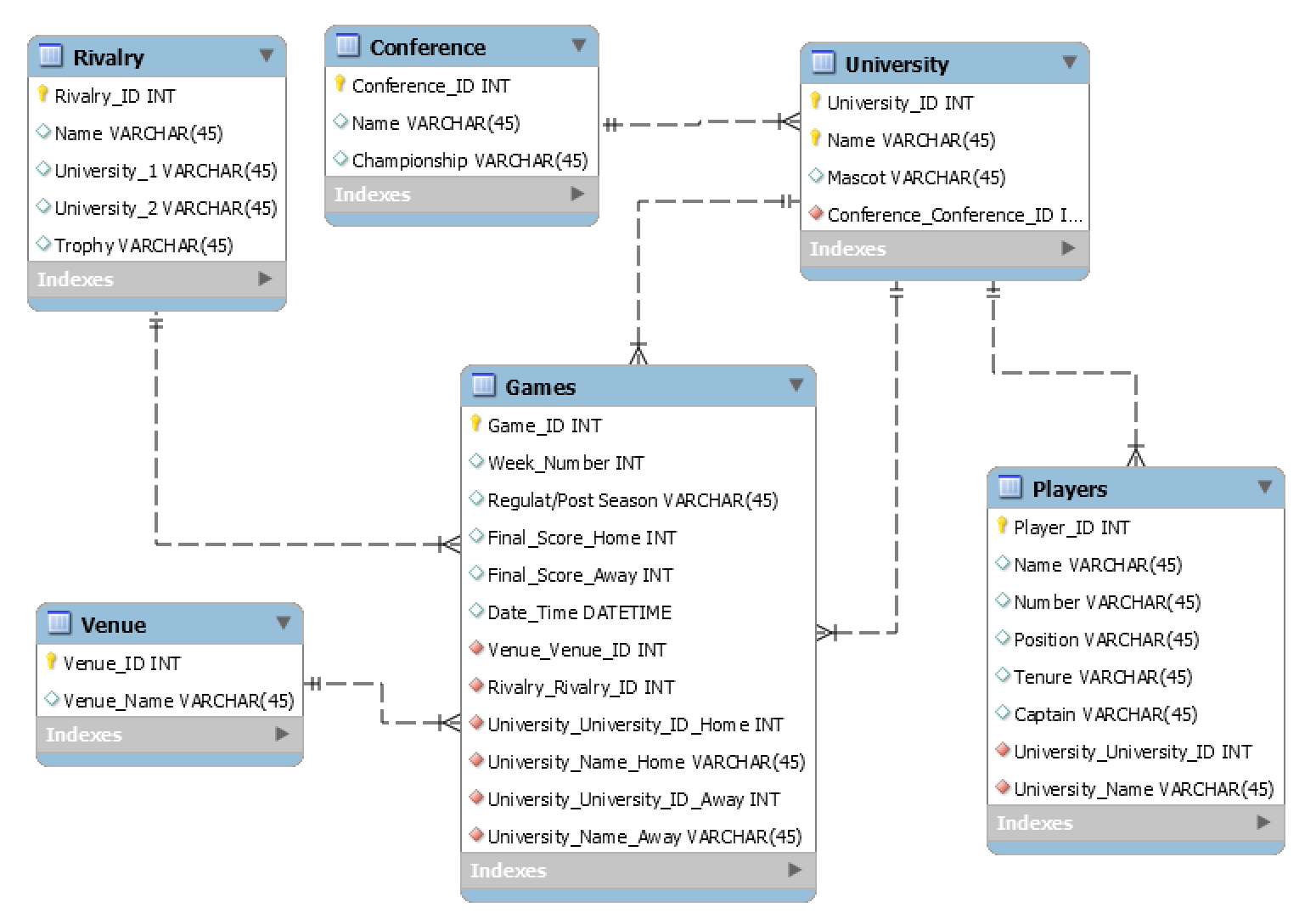


* + Capture the scores for each game



**Note:**

* Your ER model answer should identify the key entities and relations between them.
* Ensure you state all your assumptions while building the model.
* Summarize your design in plain English sentences along with the ER model.
* Insert the picture of the ER model inline with your answer (you are free to use MySQL Workbench, visio, ppt, word or any other tool to create a readable and legible ER model).



ER model includes the following tables [Rivalry], [Conference], [University], [Venue], [Games], and [Players]. [Games] table has multiple foreign keys within it which allow it to build relations with the other tables.

[Games.Venue\_Venue\_ID] establishes the 1-to-n relationship with [Vanue.Venue\_ID].

[Games.Rivalry\_ID] establishes a 1-to-n with [Rivalry.Rivalry\_ID].

There are 2 relationships between [Games] and [University] in order to allow for a Home and Away team. [Conference] has a relation to [University] through the Conference\_ID. [Players] has a relationship with [University] through the University\_ID and University\_Name.

# Question (20 Points)

In the context of the EcommerceDB we built, some customers have been complaining about the quality of the items they bought. They want to be able to return orders and get a full refund. The business has decided to support returns with the following requirements.

* Customers can return the entire order or just a part of it.
* The business must be able to track the items being returned with their quantities.
* The system must be able to capture the refunds issued on the returns with the payment methods on which they are issued.
* To prevent fraud the business team
  + wants to be able to track returns against the original order the customers bought
  + ensure only the items that were originally bought are returned
  + ensure the returned quantity does not exceed the original quantity bought

Your task is to update the design of the EcommerceDB schema to handle these new requirements.

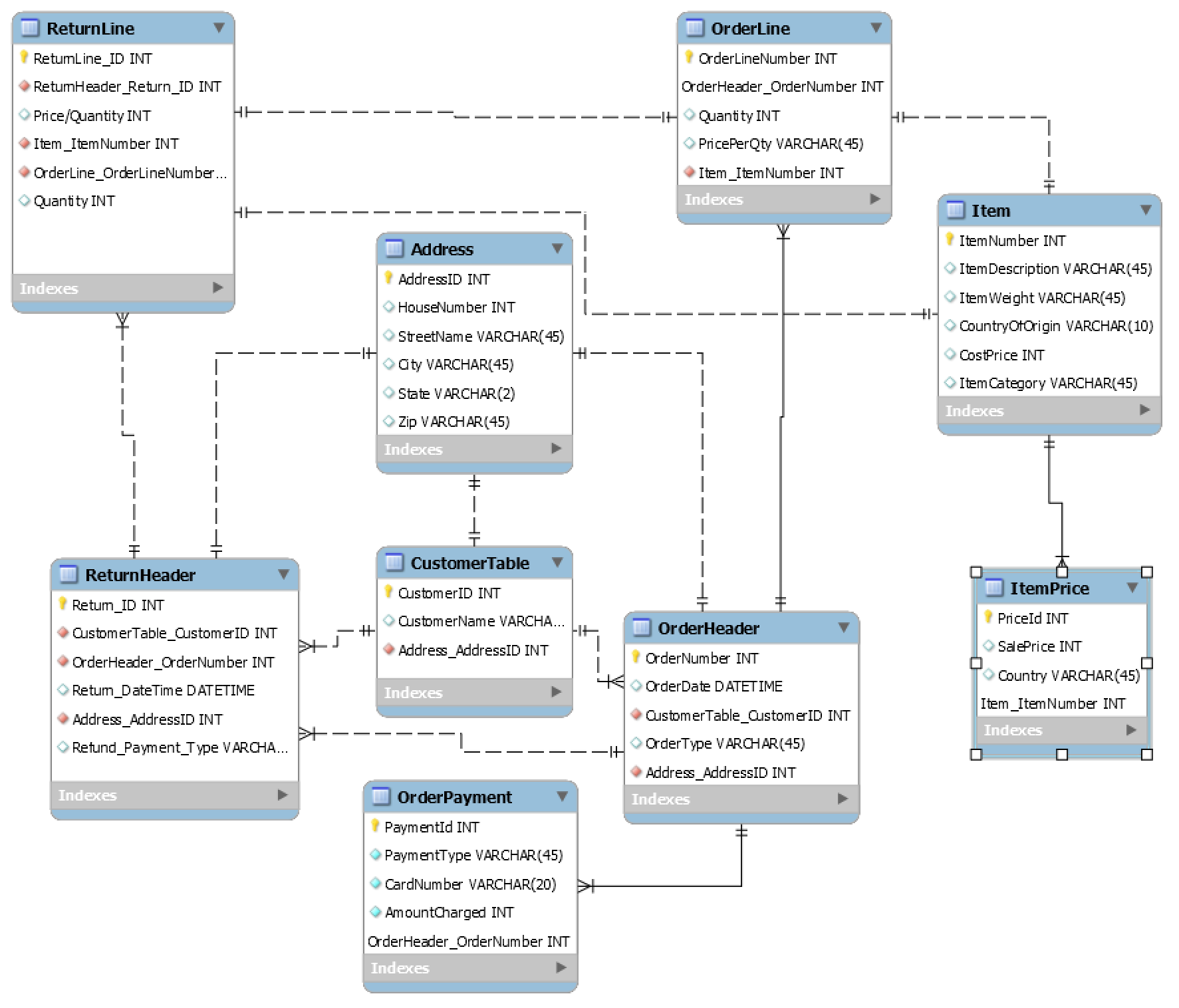
What features/capabilities of databases would you use to handle quantity validations?

Describe how your changes impact the existing orders already in the systems.

Change is the only constant. What are some of the strategies you recommend to future proof your design?

**Note:**

* Identify the relevant entities and relations that are to be designed/ enhanced from existing structure for supporting the new requirements
* State your assumptions clearly
* Summarize your design approach in a well written format in addition to the ER diagram representing any new entities.
* Insert the picture of your solution ER model inline with your answer (you are free to use MySQL Workbench, visio, ppt, word or any other tool create to a readable/legible ER model).
* You can choose to hand create sample return order record(s) to aid in better explanation of your model. The data can be mocked up in excel / notepad to better explain your design.



In this design, we add two new tables to the schema, which are “ReturnHeader” and “ReturnLine.” Each one is designed to behave similarly to their counterparts, OrderHeader and OderLine in that a return may have multiple items being returned under a single transaction. The tables are joined to multiple other tables for integrity purposes. Since we want the item being returned to match exactly the item purchased, the ReturnLine table sources the Item\_ItemNumber as a foreign key to ensure the SKU is included in the database. Furthermore, by linking to the OrderHeader table through the ReturnHeader, we are able to establish rules in place that would prevent the item being returned more than the times purchased. This could be implemented through views in the point-of-sale software which would prevent Customer\_A from returning n+1 items when they only purchased n by prompting an error. The downside to this element is that an OrderNumber would be required to validate a purchase occurred. In the event someone is gifted one of our products and they want to return it, this current model would not allow for that. In order to prepare for expansion and future requests, it would be wise to modify the schema much more and add options to be able to return items without an ordernumber (receipt). This could be done by removing the dependency on the OrderNumber and allowing for such transactions to occur. A common practice at major retailers is to allow only gift card/store credit in those types of transactions. This still may not prevent a person from buying Item\_1 at Walmart for $9.99 (on sale) and returning it at our store, which is retailed for $19.99 and getting store credit. This is a common problems most major retailers face and Amazon, like Best Buy, has begun to tackle this problem by revoking a customer’s ability to return a product whatsoever. Whether an order number is available or not, they log the return against the customer’s purchase history. Because most carriers now require legal ID to return a product, they are able to log returns to the individual. Adding that ability to this table would allow the fraud department to get ahold of the curb and prepare for future cost saving strategies. Furthermore, adding greater detail for precision analytics would also help. Including item dimensions, weight, shipping cost, delivery lag, etc. are all examples in which a company would be able to asses whether their items are being shipped in the most efficient manner possible. Information differentiating between billing address and shipping address would also provide greater insight into where the products are going and which ones are most popular where. This allows for targeted marketing, which makes more efficient and effective marketing strategy. Capturing more information on the customer would also allow them greater insight into who the customer is. This would be information such as demographics and so on. Once again, this would allow for better analytics into which customers are buying which products and also help them tailor products for specific demographics. Another example would be in the event third-party sellers are allowed to sell on the website, which has become a common trade in recent years. Allowing for that data to be captured, filtered, and shared with the seller would also require some modification. By adding these new elements, we would have clearly missed any previous returns processed since we wouldn’t have any electronic data. This would make validating those transactions from a fraud perspective a near impossible task.

# Question (20 Points)

In the context of the EcommerceDB we built, new regulations require that sales taxes be collected in the tax authority of the customer. The tax authority is made up of the state, the county, and the city. Each of these is an independent authority and changes their tax rates independently. The business must support the collection of taxes for all of these tax authorities.

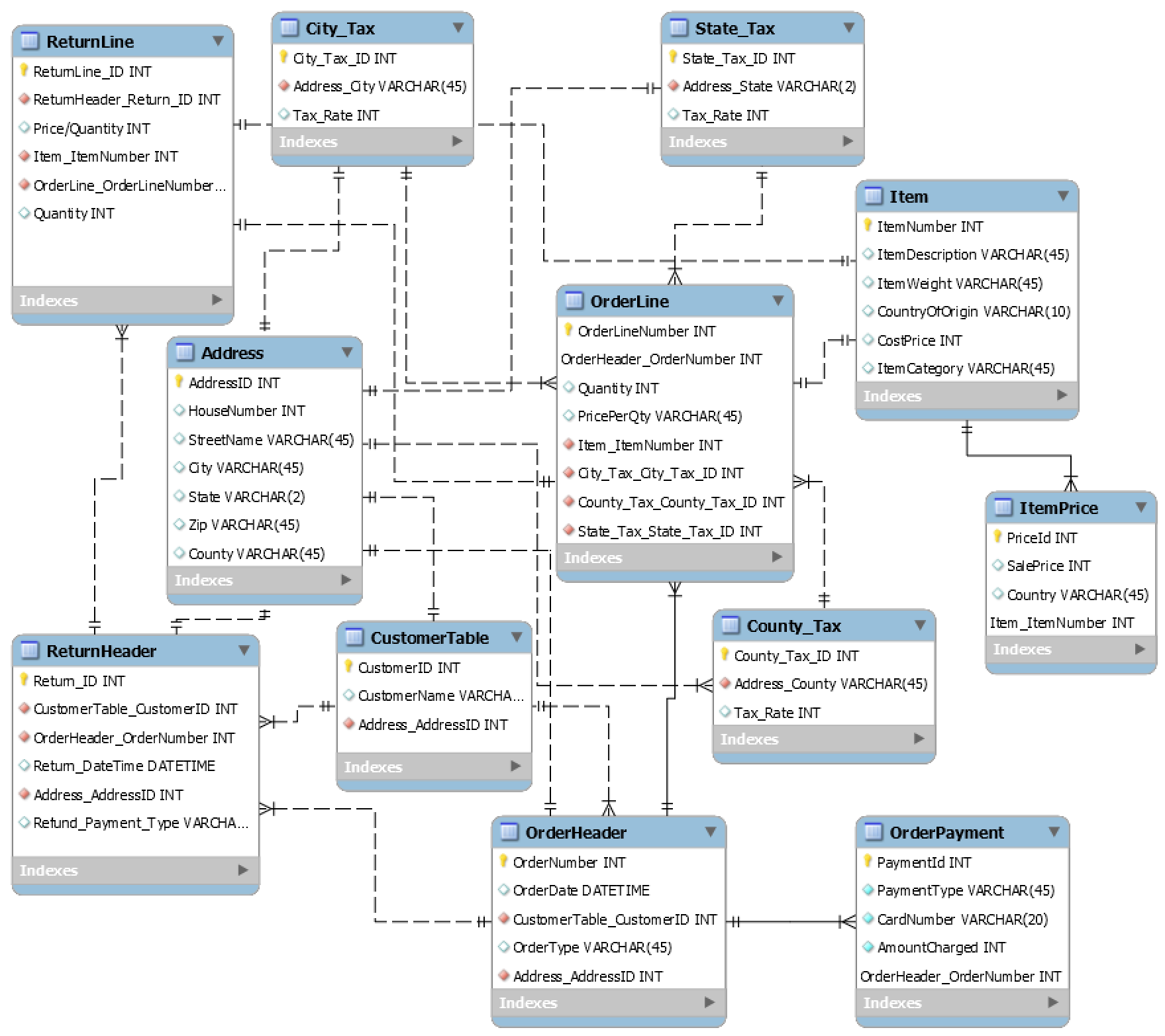
Your task is to update the design of the EcommerceDB schema to handle these new requirements.

Describe how your changes impact the existing orders already in the systems.

Change is the only constant. What are some of the strategies you recommend to future proof your design?

**Note:**

* Identify the relevant entities and relations that are to be designed/ enhanced from existing structure for supporting the new requirements
* State your assumptions clearly
* Summarize your design approach in a well written format in addition to the ER diagram representing any new entities.
* Insert the picture of your solution ER model inline with your answer (you are free to use MySQL Workbench, visio, ppt, word or any other tool create to a readable/legible ER model).
* You can choose to hand create sample return order record(s) to aid in better explanation of your model. The data can be mocked up in excel / notepad to better explain your design.

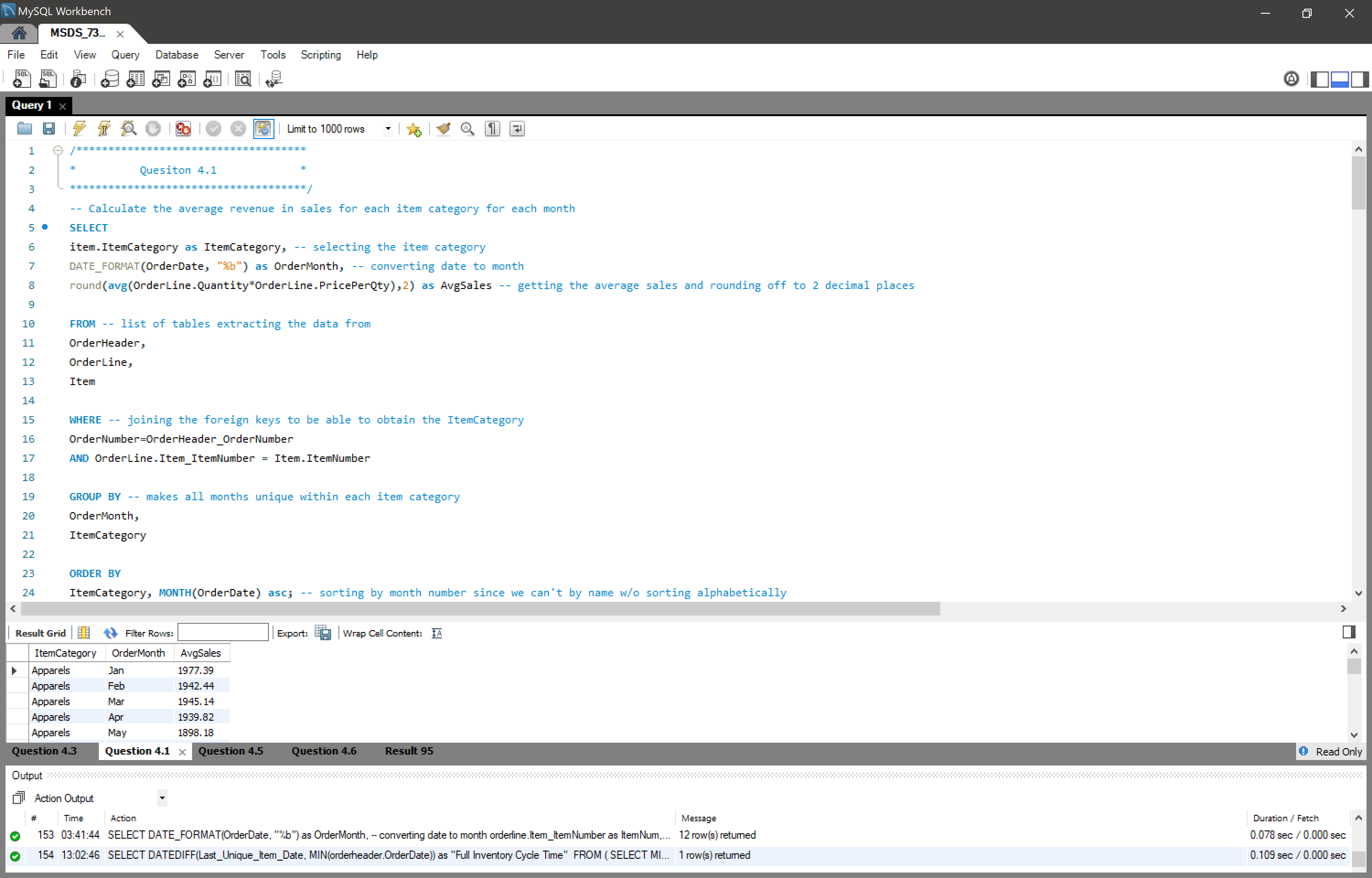


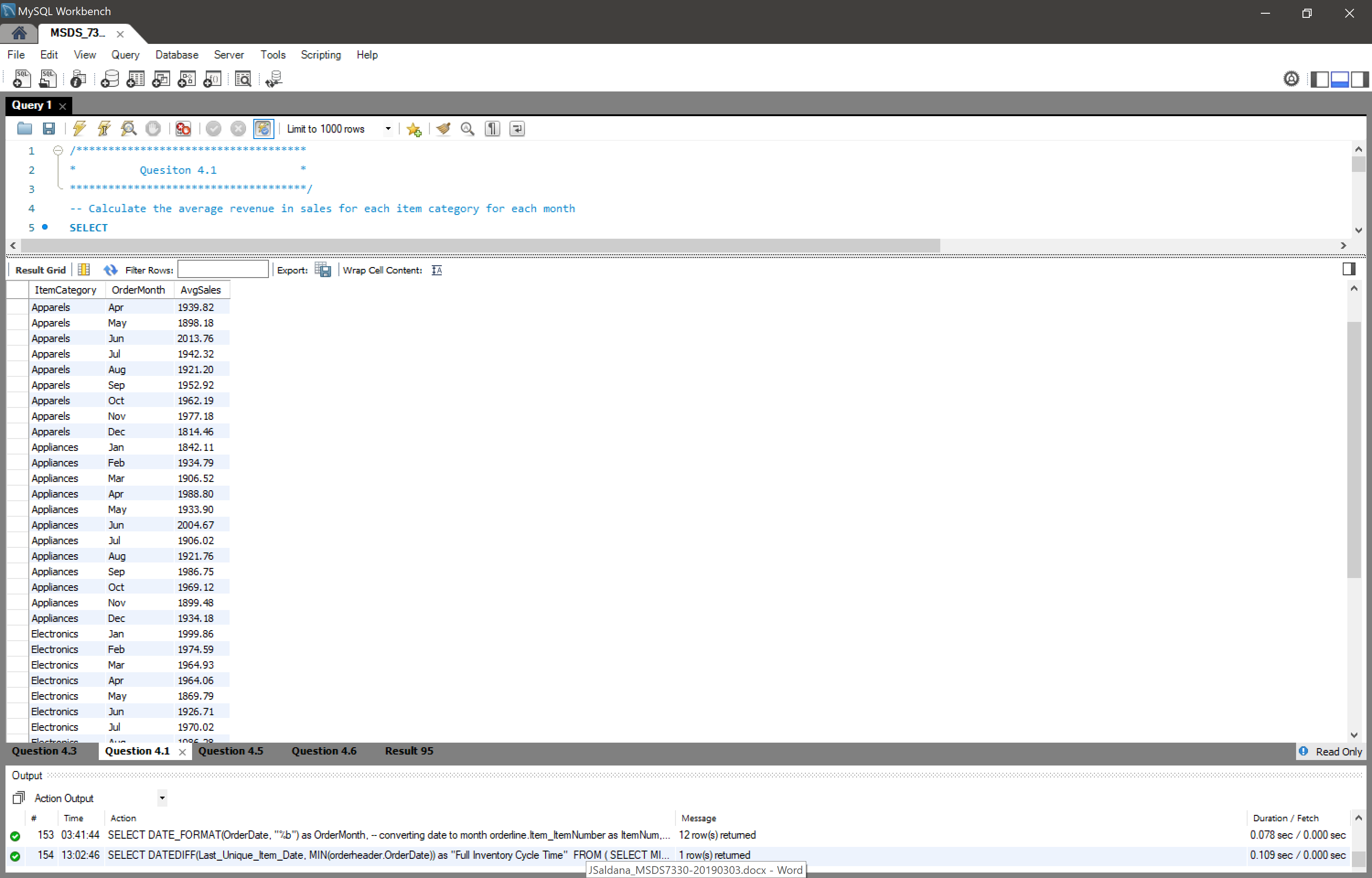
This design assumes the same assumptions as the schema from Q2 and builds on that design to now incorporate local, county, and state taxes. The changes made include the addition of 4 tables: City\_Tax, County\_Tax, and State\_Tax. The city, county, and state tables include their respected foreign key from the Address table (since the billing and shipping address are assumed to be the same). Considering each entity has its own tax rate and there is overlap amongst each, it was important to separate them into their own tables. This is the same reason why I had to add County to the Address table. There are several zip codes that over lap cities, counties, and even state lines. The same applies to cities, which may extend over to different counties and even states. Though this design, the tax rate for the customer would be calculated before a purchase is made since the determining factors are foreign keys from the address table. The State\_Tax table would include a list of every state and its respected sales tax. The County\_Tax would include every county in the US and its respected sales tax. The same would be said about the city table. This also allows for easily manipulation in the event a tax rate needs to be modified. These tax rates IDs are then inserted as foreign keys into the OrderLine table for further calculations if need be. To further proof the table for tax purposes, It would be recommended to include foreign entities and their respected tax rates and government entities. Along with inclusion of foreign government entities, a currency selection would need to be included since tax rates from England are based on pounds and not USD. The assumptions are that the company only allows for domestic shipping in the US and that the billing address and the shipping address are the same address. Furthermore, it also assumes all customers are US based and doesn’t allow for international customers.

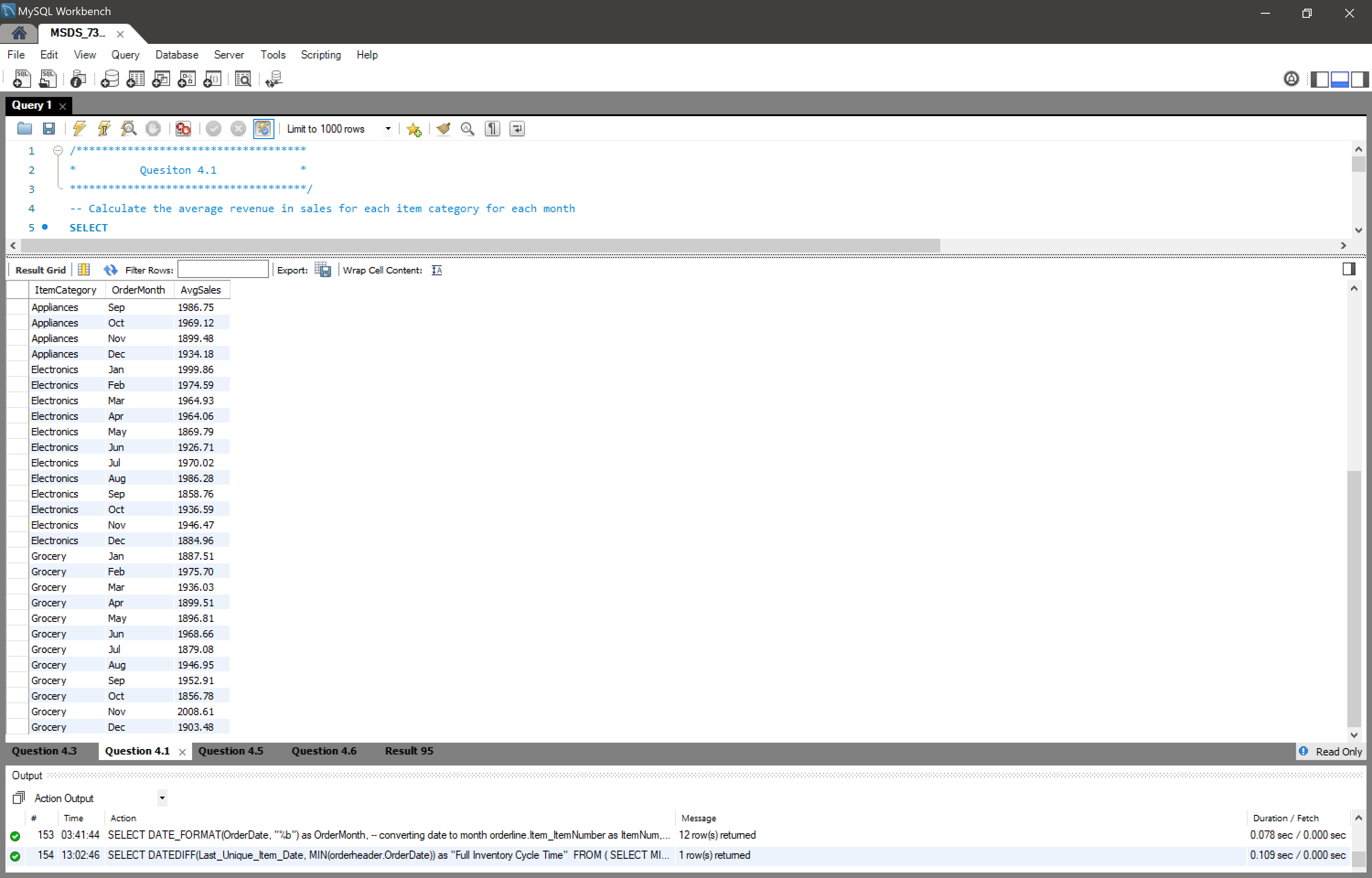
# Question (30 Points)

In the context of the Ecommerce Order Database we designed in the class, create SQL queries to address the following questions. Turn in your documented and explained SQL queries and screenshots of your SQL queries on the Ecommerce database.

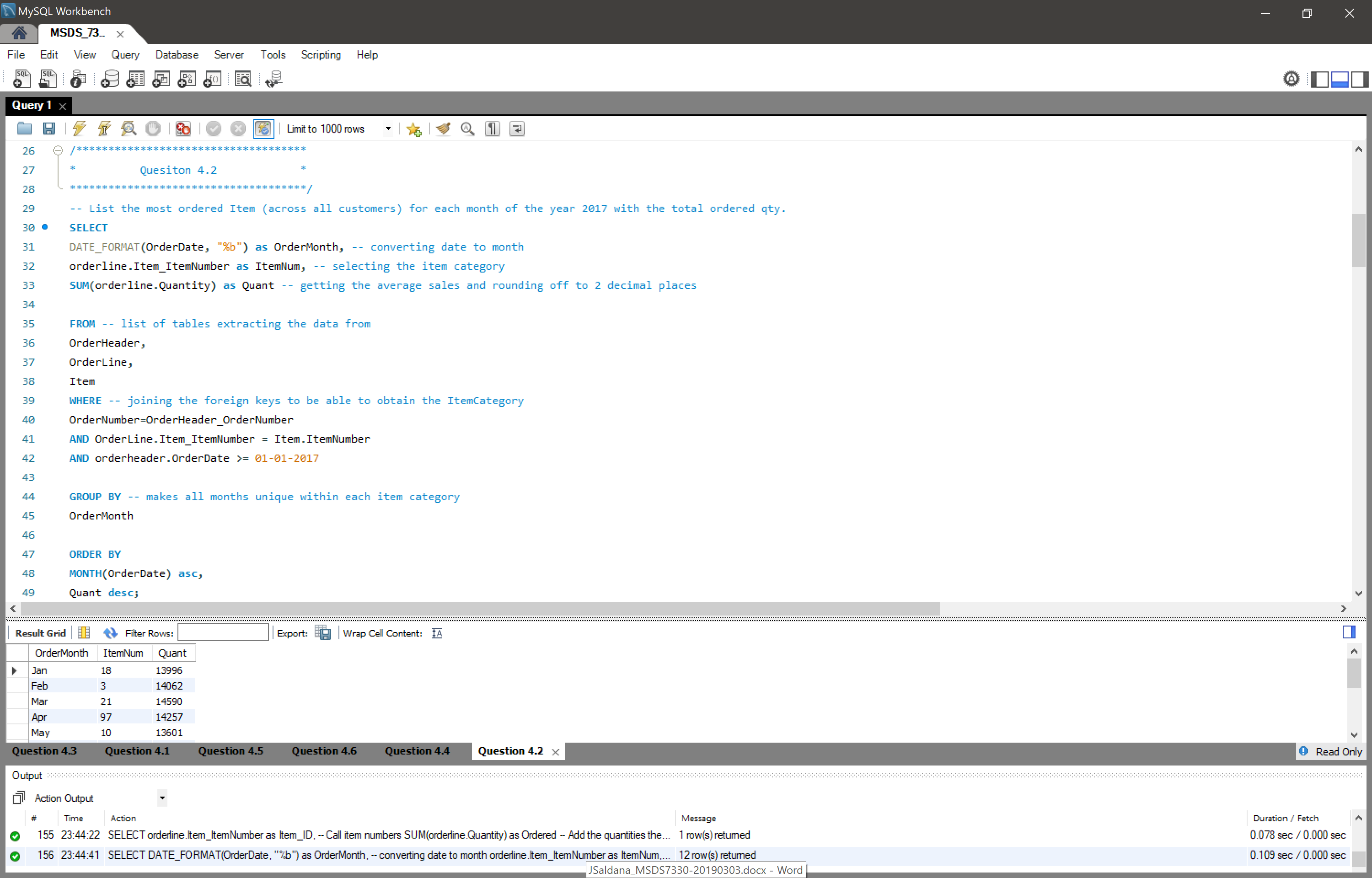


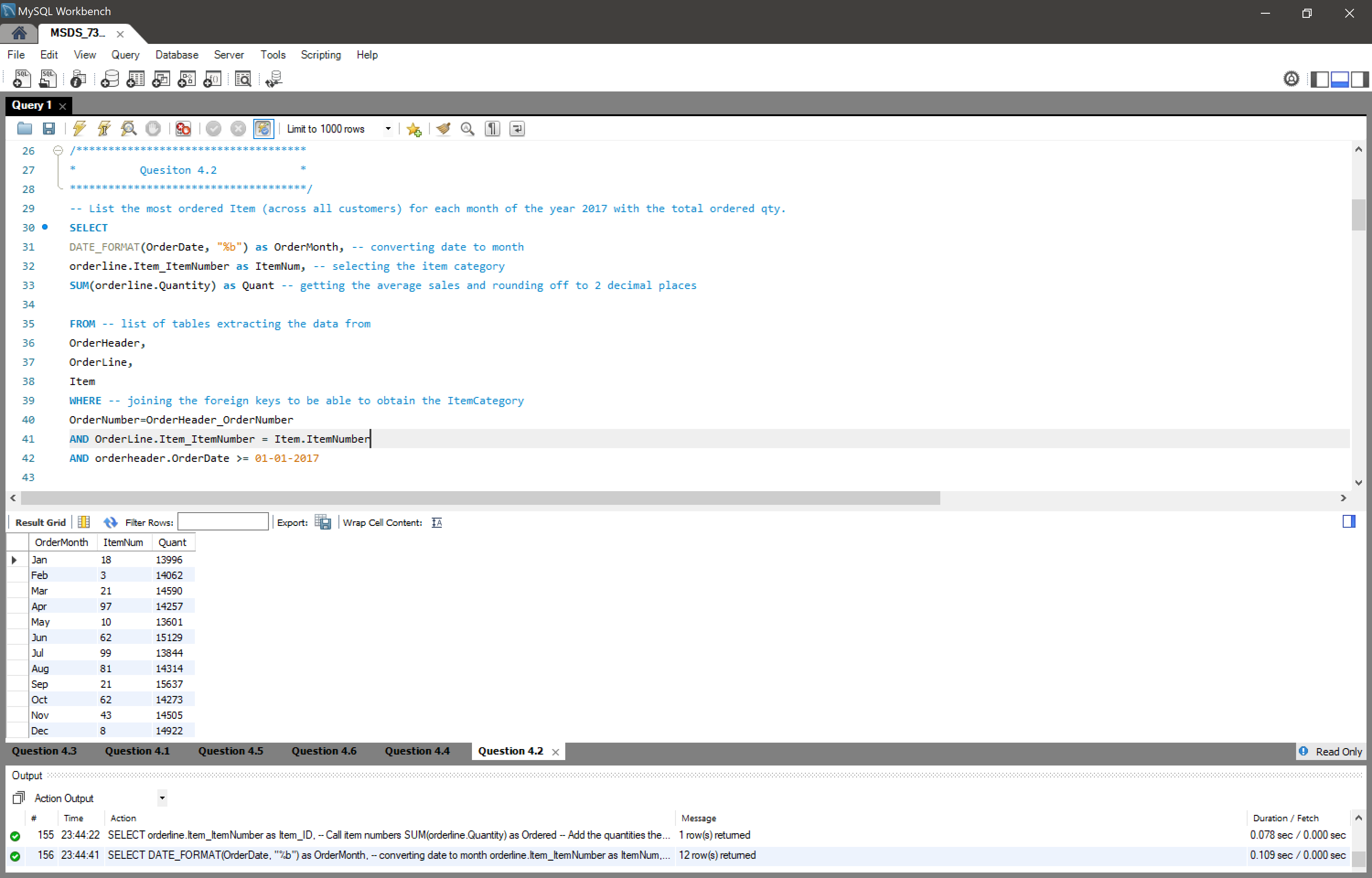




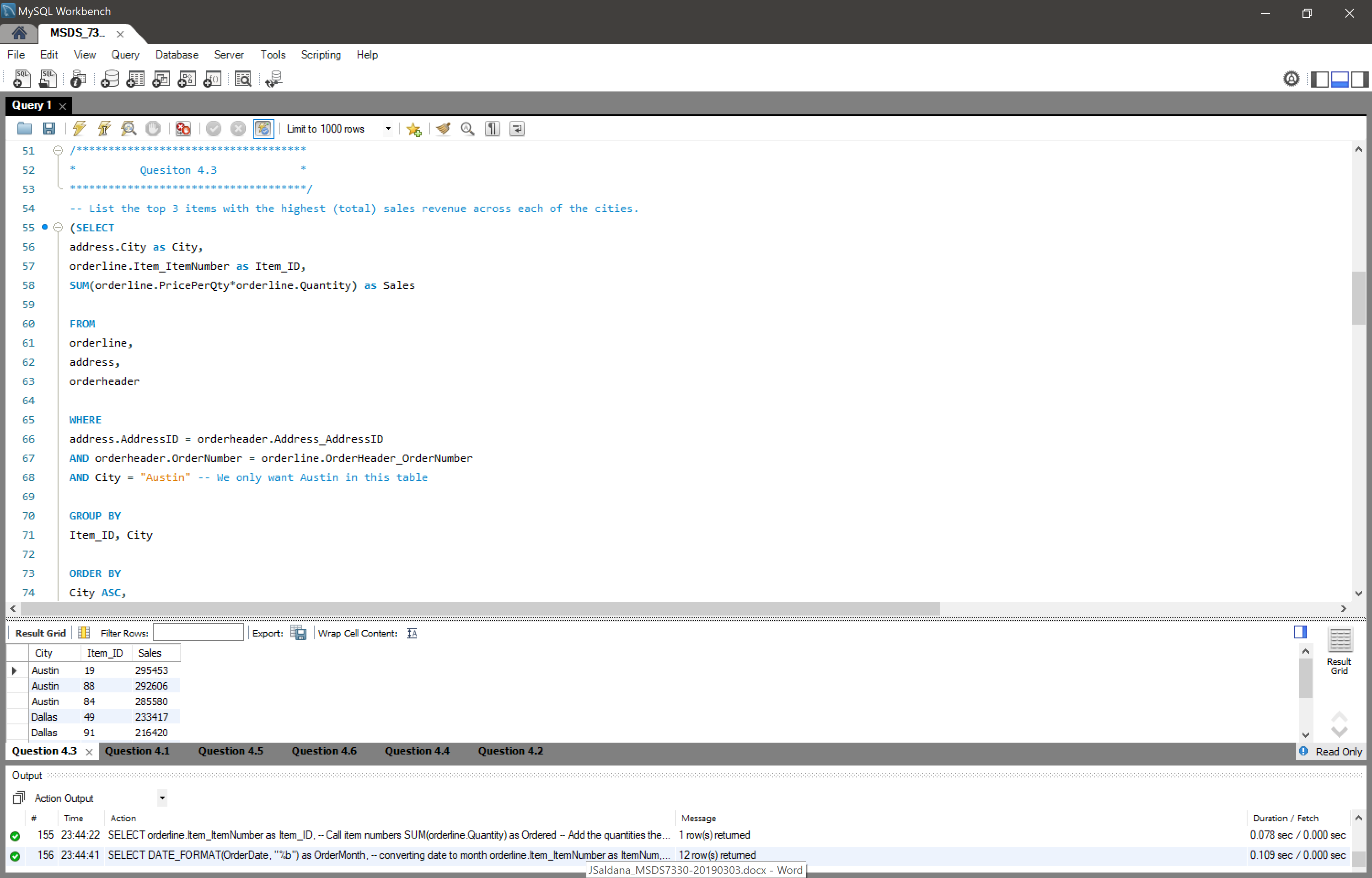


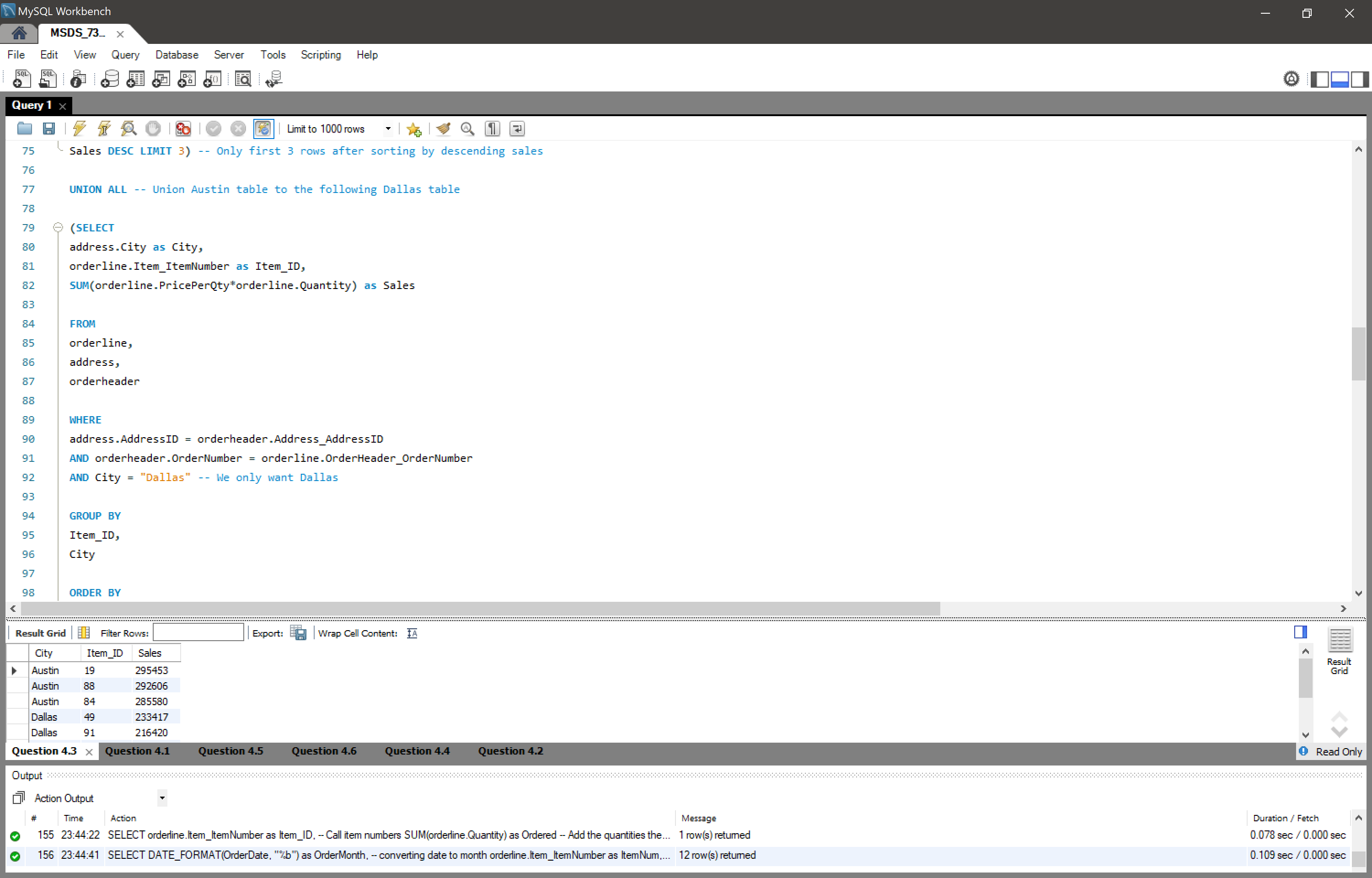


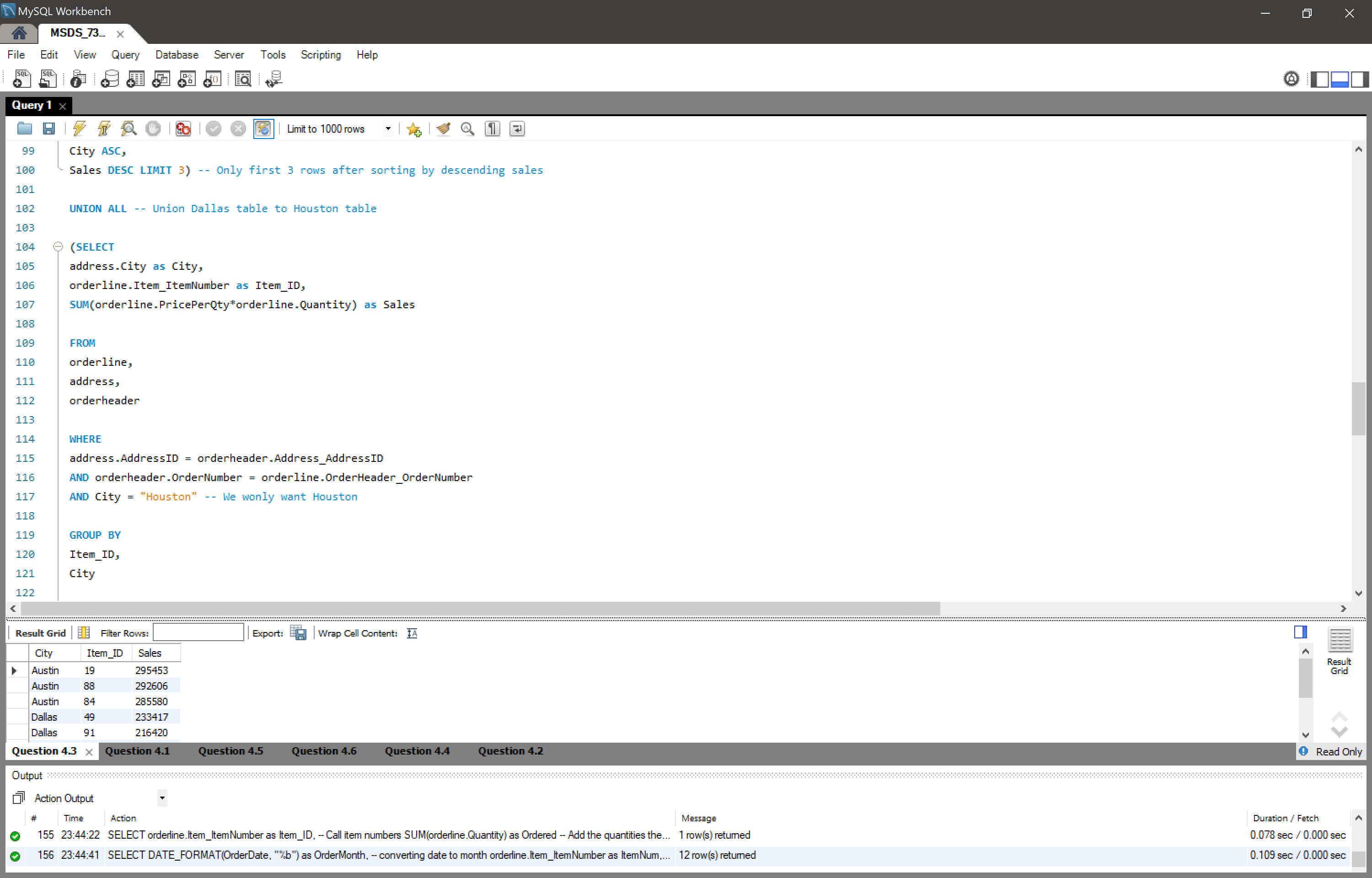


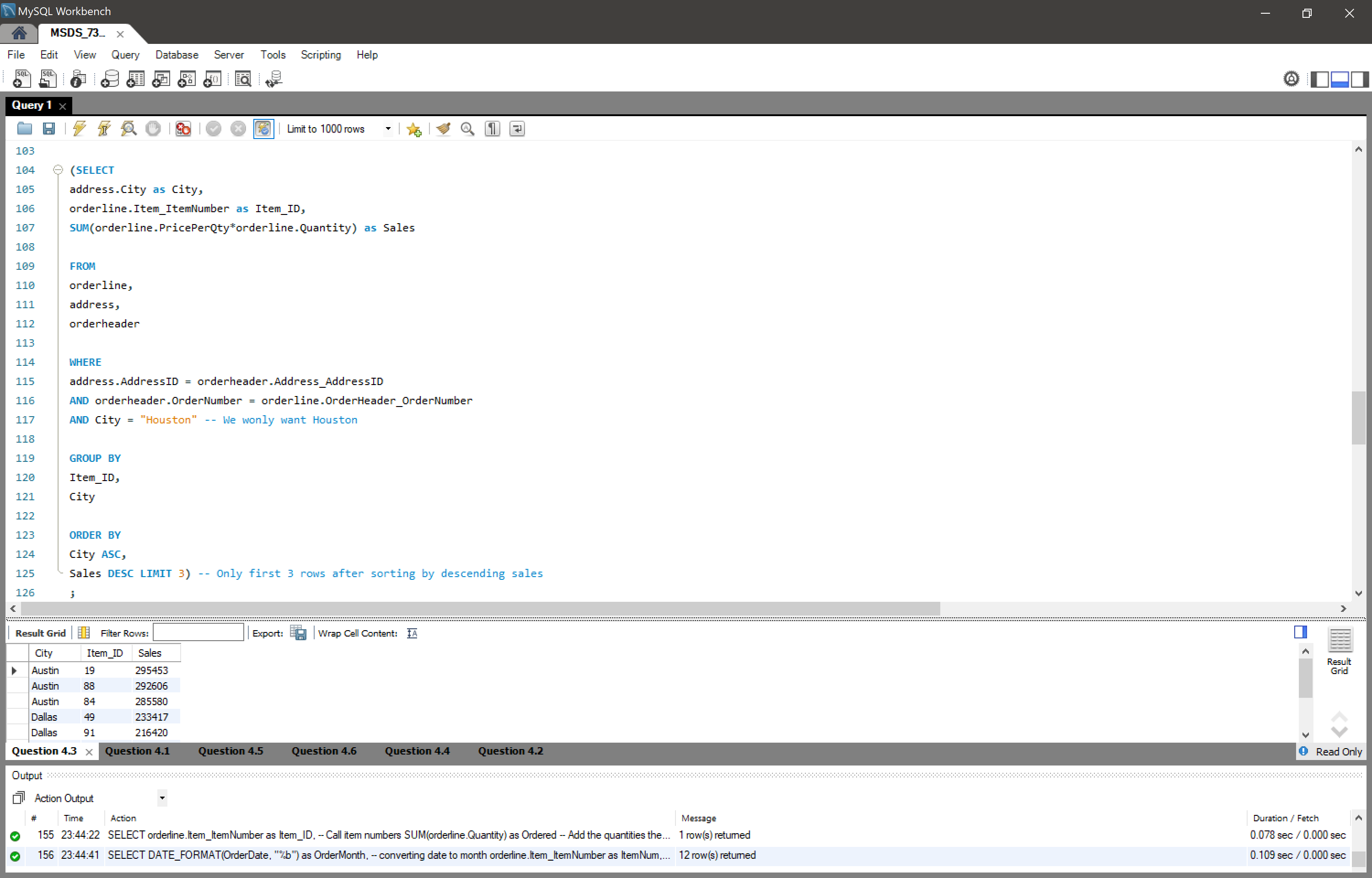




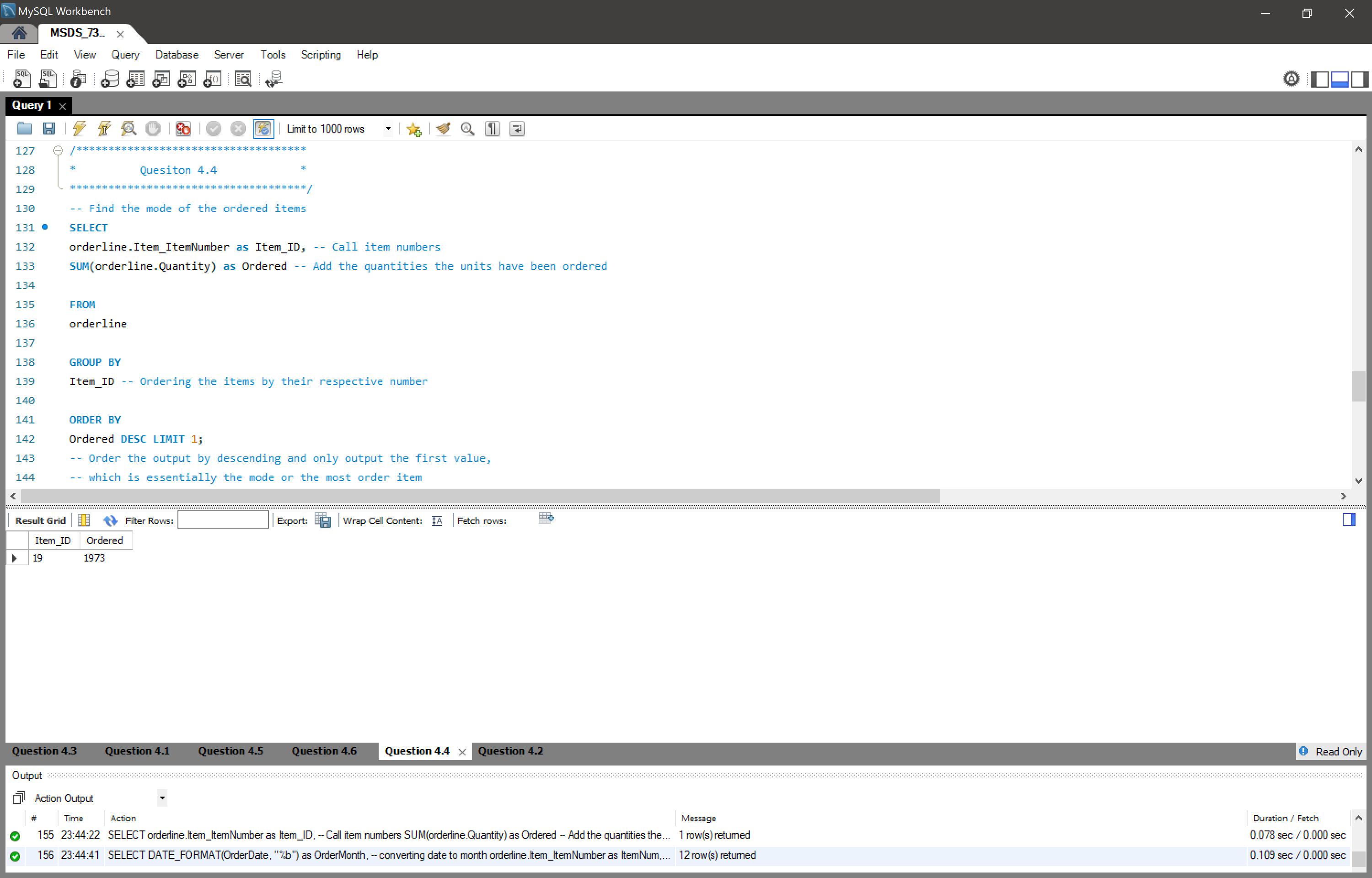




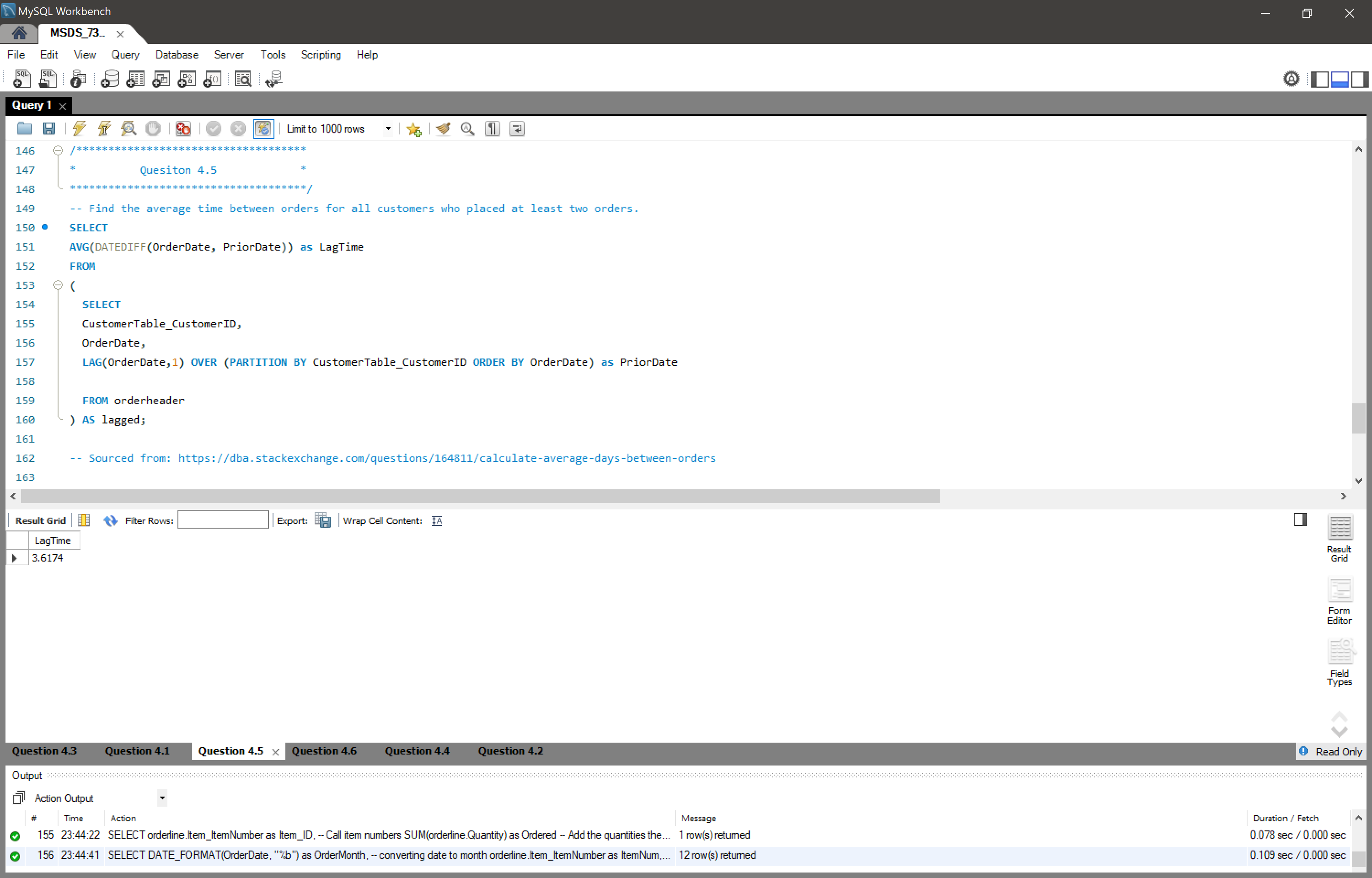




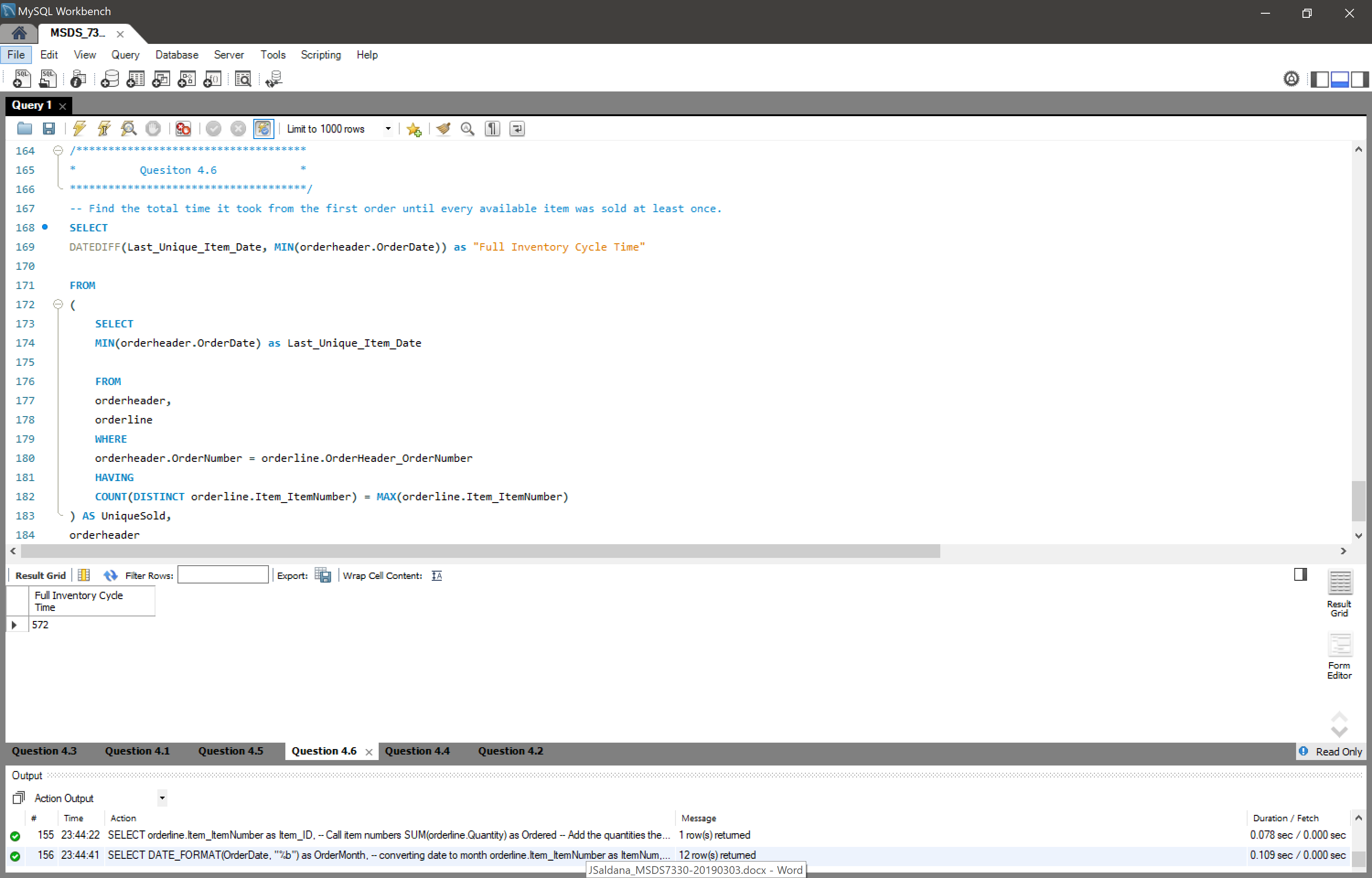












# Question (10 Points)

Define each of the ACID properties and in 2-3 paragraphs for each property explain how a relational database maintains that property and why that property is important in relational databases.

**Atomicity**

*Atomicity refers to the effects of a transaction in which it must either be fully committed or aborted. Executing partially committed transactions would mean the database lacks atomicity.*

For relational databases, maintaining atomicity is critical. Consider a relational database for a bank in which a customer has a checking account with a balance of $1,000 and a savings account with a balance of $100. Now, say this customer wishes to transfer $500 from the checking account to the savings account. The customer submits the transfer request (transaction) and the system withdraws the funds from the account but fails to add them to savings since the transaction failed. Now the balance of the customer is $500 in the checking and $100 in the savings since the other $500 have disappeared.

At this point, the database is no longer atomic since it failed to abort the transaction fully. This is a just a small example in which lack of atomicity would lead to negative (and inaccurate) results. Considering how many transactions are processed daily by a large organization, failure to adhere to the atomicity principle would result in lost data and a very expensive project to correct the issues that would have occurred. The problem is that most of the time, these types of problems wouldn’t be noticed unless the effects were negative and impactful.

**Consistency**

*Consistency guarantees the database will remain in a consistent state after a transaction is executed.*

In relational database management systems (RDMS), consistency is another important aspect of database integrity. As databases get larger and complex, the number of transactions executed increase. Consistency protects the integrity of the data by ensuring the transaction doesn’t alter the database in any other methods that are not allowed. In the same scenario as our previous example, if the customer goes deposits $100 into his checking account, the customer would expect his balance to increase by $100. A transaction listed as a deposit (or more formally a credit in the financial sector) *should* add to the final balance of the account. If the transaction deducts from the customer’s account, then the database is no longer in a consistent state.

For relational database, this would alter the given table along with all others who rely on that given variable which has been altered incorrectly. Say the transaction is executed incorrectly by instead deducting (also known as debiting) the checking account. The customer would then attempt to make a purchase using that money the customer suspects is in the account. Assuming the sum of the available funds is used when a debit is posted to the account such as a purchase at the store, if the database is not in a consistent state then the transaction will decline the transaction since it will not show the funds are available in the account.

Its important to note that consistency doesn’t validate the accuracy of the data or ensure it records correctly. Consistency simply ensures the transactions follow the proper rules in place. There are instances where consistency may be lost such as when there are concurring transactions being executed. In this case, it is important for the database to have measures in place such as concurrency-control schemes to handle these transactions.

**Isolation**

*Isolation ensures the transactions being processed are isolated from each other so that other transactions are not being executed concurrently.*

For a relational database, this ties into the scenario mentioned above in which a customer may submit multiple transactions at the same time. Say the customer is making a cash deposit of $100 into his account early in the morning. At the same exact time, his scheduled auto-payment for his utility bills are being processed. A database without the isolation principle would process both transactions at the same time. This would mean that his account would now his account would have 2 balances, one where the bill has been paid and a deposit has not been made and another where the deposit has been made but no bill has been paid.

In a relational database, this could cause major headaches since the available balance would likely return an incorrect value or an error since there are now two balances. As the scale of the database grows, so does the need to maintain isolation over all transactions. An ideal method of handling the transactions would be through scheduling. This would give the database the ability to receive simultaneous transactions but put them aside and schedule them to handle then in isolation. For the end user (customer), this may be seen as “Funds will be available the following business day.” The database is able to break up and create multiple schedules as they come and be able to commit them accordingly.

**Durability**

Durability requires that the history of the transactions that have been committed are not lost, even in the event of a system/power failure.

Servers hosting the database may be shut down for multiple reasons (power outage, updates, etc.). During times in which the host is not available or any other event, the database must be able to maintain the data of the transactions. In line with our customer and bank example, say the customer makes his deposit and leaves the bank assuming the funds will be there the following business day. The system schedules his transaction to occur at its respective time but there is a scheduled update to the database in which the servers will be taken offline. The transaction is not processed in time before the update and is lost. Once the system comes online, there is no record of the transaction ever occurring and when the customer calls to complain there is no record of him ever even attempting to make a deposit. As you could imagine, this would create a legal issue among others for the bank.

Under a relational database, this would clearly alter the final balances for customers but also scheduled payments being made to institutions in which the customers used their debit/credit cards. The bank would have no record and all that money would be lost, clearly resulting in a major financial loss. Maintaining durability ensures this does not take place. Methods in place to maintain durability are such as creating logs of the transactions and current states of the database to ensure there is a historical record in which the data may be reverted. Yet, what happens when the system is turned off while a transaction is taking place? In this instance, the atomicity of the database would recognize the failed transaction and abort it. The system would then have the ability to either restart the transaction or kill it once it comes online again. Clearly, if the transaction is killed, there would be a recorded log of such an even and the system would be able to revert back to the data and re-attempt to do so if need be.