SQL Assignments

SQL related assignments will be on Wide World Importers Database if not otherwise introduced.

1. List of Persons’ full name, all their fax and phone numbers, as well as the phone number and fax of the company they are working for (if any).

with cte as

(select s.PrimaryContactPersonID as id, s.PhoneNumber, s.FaxNumber

from Purchasing.Suppliers s

where s.PrimaryContactPersonID is not null

union

select s.AlternateContactPersonID as id, s.PhoneNumber, s.FaxNumber

from Purchasing.Suppliers s

where s.AlternateContactPersonID is not null

union

select c.PrimaryContactPersonID as id, c.PhoneNumber, c.FaxNumber

from Sales.Customers c

where c.PrimaryContactPersonID is not null

union

select c.AlternateContactPersonID as id, c.PhoneNumber, c.FaxNumber

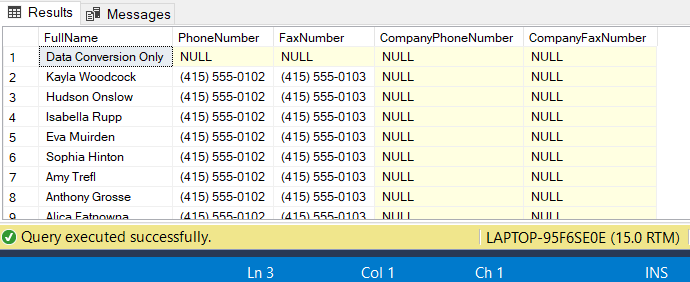
from Sales.Customers c

where c.AlternateContactPersonID is not null

)

select p.FullName, p.PhoneNumber, p.FaxNumber, cte.PhoneNumber as CompanyPhoneNumber, cte.FaxNumber as CompanyFaxNumber

from Application.People p left join cte on p.PersonID = cte.id

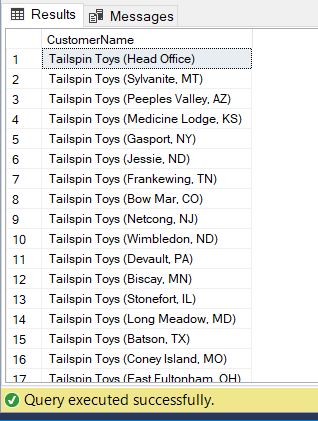


1. If the customer's primary contact person has the same phone number as the customer’s phone number, list the customer companies.

select c.CustomerName

from Sales.Customers c left join Application.People p on c.PrimaryContactPersonID = p.PersonID

where c.PhoneNumber = p.PhoneNumber



1. List of customers to whom we made a sale prior to 2016 but no sale since 2016-01-01.

select c.CustomerName

from Sales.Customers c

where c.CustomerID not in

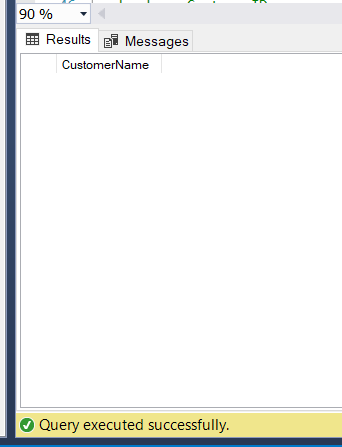
(

select o.CustomerID

from Sales.Orders o

where o.OrderDate >= cast('2016-01-01' as date)

)



1. List of Stock Items and total quantity for each stock item in Purchase Orders in Year 2013.

select temp.StockItemID, s.StockItemName, temp.OrderedOuters \* s.QuantityPerOuter as TotalQuantity

from

(

select ol.StockItemID, sum(ol.OrderedOuters) as OrderedOuters

from Purchasing.PurchaseOrderLines ol join Purchasing .PurchaseOrders o on ol.PurchaseOrderID = o.PurchaseOrderID

where datepart(year, o.OrderDate) = 2013

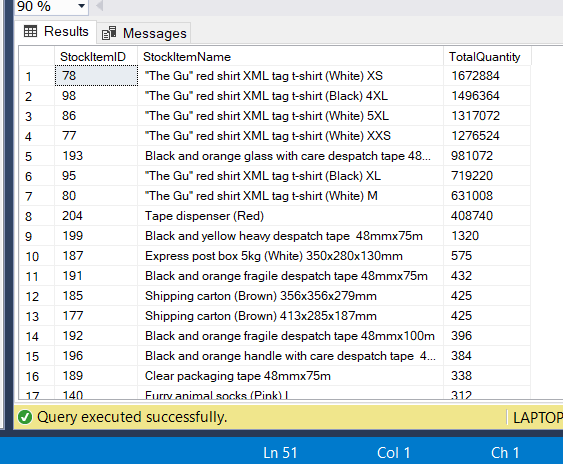
group by ol.StockItemID

) temp

left join Warehouse.StockItems s

on temp.StockItemID = s.StockItemID

order by TotalQuantity desc



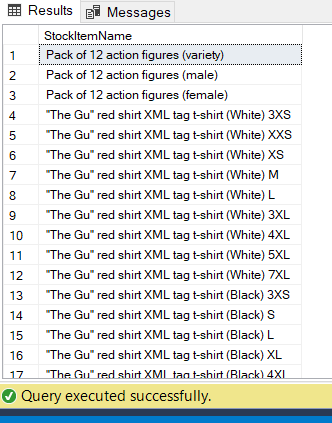
1. List of stock items that have at least 10 characters in description.

select s.StockItemName

from Warehouse.StockItems s left join Purchasing.PurchaseOrderLines ol

on s.StockItemID = ol.StockItemID

where len(ol.Description) >= 10



1. List of stock items that are not sold to the state of Alabama and Georgia in 2014.

select distinct s.StockItemName

from Warehouse.StockItems s left join Sales.OrderLines ol on s.StockItemID = ol.StockItemID

left join Sales.Orders o on ol.OrderID = o.OrderID

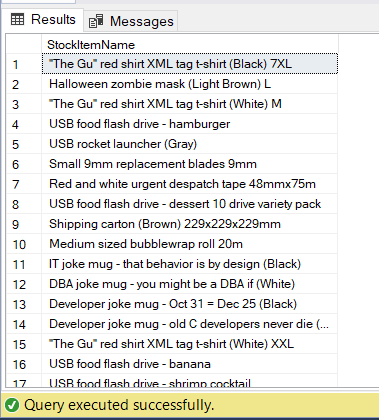
left join Sales.Customers c on o.CustomerID = c.CustomerID

left join Application.Cities ct on ct.CityID = c.DeliveryCityID

left join Application.StateProvinces sp on sp.StateProvinceID = ct.StateProvinceID

where datepart(year, o.OrderDate) = 2015 and

sp.StateProvinceName not in('Alabama','Geogia')



1. List of States and Avg dates for processing (confirmed delivery date – order date).

select sp.StateProvinceName, temp.avg\_ProcessDates

from

(

select ct.StateProvinceID, avg(datediff(day,o.OrderDate,i.ConfirmedDeliveryTime))as avg\_ProcessDates

from Sales.Orders o left join Sales.Invoices i on o.OrderID = i.OrderID

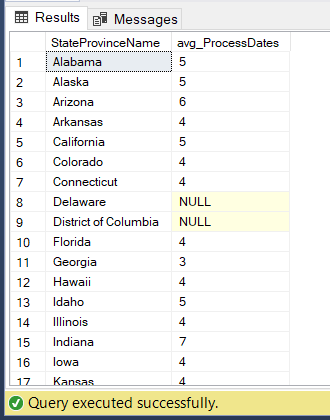
left join Sales.Customers c on o.CustomerID = c.CustomerID

left join Application.Cities ct on ct.CityID = c.DeliveryCityID

group by ct.StateProvinceID

) temp

right join Application.StateProvinces sp on sp.StateProvinceID = temp.StateProvinceID



1. List of States and Avg dates for processing (confirmed delivery date – order date) by month.

select sp.StateProvinceName, temp.avg\_ProcessDates

from

(

select ct.StateProvinceID, avg(datediff(day,o.OrderDate,i.ConfirmedDeliveryTime))as avg\_ProcessDates

from Sales.Orders o left join Sales.Invoices i on o.OrderID = i.OrderID

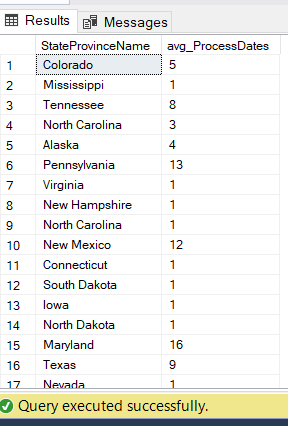
left join Sales.Customers c on o.CustomerID = c.CustomerID

left join Application.Cities ct on ct.CityID = c.DeliveryCityID

group by ct.StateProvinceID, datepart(year, o.OrderDate), datepart(month, o.OrderDate)

) temp

right join Application.StateProvinces sp on sp.StateProvinceID = temp.StateProvinceID



1. List of StockItems that the company purchased more than sold in the year of 2015.

with temp\_pur as

(

select pol.StockItemID, s.StockItemName, sum(pol.OrderedOuters \* s.QuantityPerOuter) as purchasequantity

from Purchasing.PurchaseOrderLines pol left join Purchasing.PurchaseOrders po on pol.PurchaseOrderID = po.PurchaseOrderID

left join Warehouse.StockItems s on pol.StockItemID = s.StockItemID

where datepart(year, po.OrderDate) = 2015

group by pol.StockItemID, s.StockItemName

)

, temp\_or as

(

select ol.StockItemID, s.StockItemName, sum(ol.Quantity) as orderquantity

from Sales.OrderLines ol left join Sales.Orders o on ol.OrderID = o.OrderID

left join Warehouse.StockItems s on ol.StockItemID = s.StockItemID

where datepart(year,o.OrderDate) = 2015

group by ol.StockItemID, s.StockItemName

)

select distinct temp\_pur.StockItemName

from temp\_pur full join temp\_or

on temp\_pur.StockItemID = temp\_or.StockItemID

where temp\_pur.purchasequantity > temp\_or.orderquantity

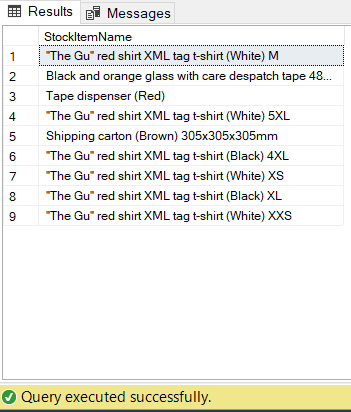
union

select distinct temp\_or.StockItemName

from temp\_pur full join temp\_or

on temp\_pur.StockItemID = temp\_or.StockItemID

where temp\_pur.purchasequantity > temp\_or.orderquantity



1. List of Customers and their phone number, together with the primary contact person’s name, to whom we did not sell more than 10 mugs (search by name) in the year 2016.

with cte as (

select o.CustomerID, sum(temp1.mug\_order\_quantity) as mug\_cus\_quantity

from

(

select ol.OrderID, sum(ol.Quantity) as mug\_order\_quantity

from Sales.OrderLines ol

where ol.Description like'%mug%'

group by ol.OrderID

) temp1

right join Sales.Orders o on temp1.OrderID = o.OrderID

where datepart(year,o.OrderDate)=2016

group by o.CustomerID

)

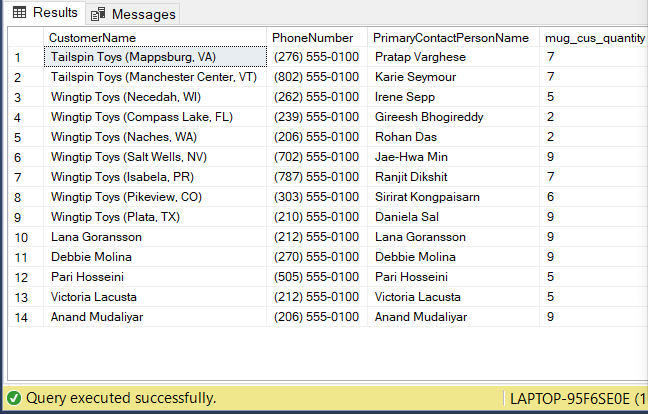
select c.CustomerName, c.PhoneNumber, p.FullName as PrimaryContactPersonName, cte.mug\_cus\_quantity

from Sales.Customers c

left join Application.People p on c.PrimaryContactPersonID = p.PersonID

left join cte on c.CustomerID = cte.CustomerID

where cte.mug\_cus\_quantity < 10



1. List all the cities that were updated after 2015-01-01.

select ct.CityName

from

(

select distinct s.DeliveryCityID

from Purchasing.Suppliers s

where datepart(year, s.ValidFrom) >= 2015

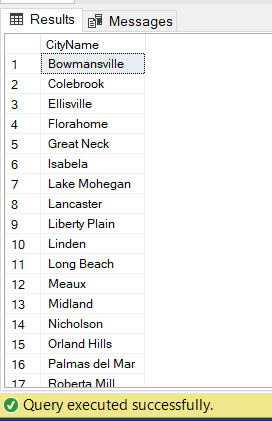
union

select distinct s.DeliveryCityID

from Sales.Customers s

where datepart(year, s.AccountOpenedDate) >= 2015

)temp join Application.Cities ct on ct.CityID = temp.DeliveryCityID



1. List all the Order Detail (Stock Item name, delivery address, delivery state, city, country, customer name, customer contact person name, customer phone, quantity) for the date of 2014-07-01. Info should be relevant to that date.

select i.StockItemName,

CONCAT(c.DeliveryAddressLine1,c.DeliveryAddressLine2) as deliveryaddress,

sp.StateProvinceName,

ct.CityName,

ctr.CountryName,

c.CustomerName,

p1.FullName as primarycontactperson,

p2.FullName as altercontactperson,

c.PhoneNumber as customerphone,

temp.quantity as quantity

from

(

select ol.OrderID, ol.StockItemID, Sum(Quantity) as quantity

from Sales.OrderLines ol left join Sales.Orders o on ol.OrderID = o.OrderID and o.OrderDate = cast('2014-07-01' as date)

group by ol.OrderID, ol.StockItemID

)temp

left join Warehouse.StockItems i on temp.StockItemID = i.StockItemID

left join Sales.Orders o on o.OrderID = temp.OrderID

left join Sales.Customers c on c.CustomerID = o.CustomerID

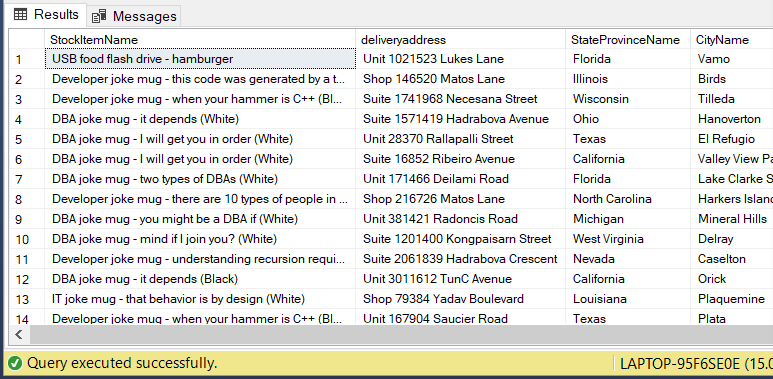
left join Application.People p1 on p1.PersonID = c.PrimaryContactPersonID

left join Application.People p2 on p2.PersonID = c.AlternateContactPersonID

left join Application.Cities ct on ct.CityID = c.DeliveryCityID

left join Application.StateProvinces sp on sp.StateProvinceID = ct.StateProvinceID

left join Application.Countries ctr on ctr.CountryID = sp.CountryID



1. List of stock item groups and total quantity purchased, total quantity sold, and the remaining stock quantity (quantity purchased – quantity sold)

select p.StockGroupName, p.total\_purchase, s.total\_sale,

p.total\_purchase - s.total\_sale as remaining\_stock

from

(

select sg.StockGroupName,

sum(pol.OrderedOuters \* i.QuantityPerOuter) as total\_purchase

from Purchasing.PurchaseOrderLines pol left join Warehouse.StockItems i on i.StockItemID = pol.StockItemID

join Warehouse.StockItemStockGroups ssg on ssg.StockItemID = pol.StockItemID

join Warehouse.StockGroups sg on sg.StockGroupID = ssg.StockGroupID

group by sg.StockGroupName

) p join

(

select sg.StockGroupName,

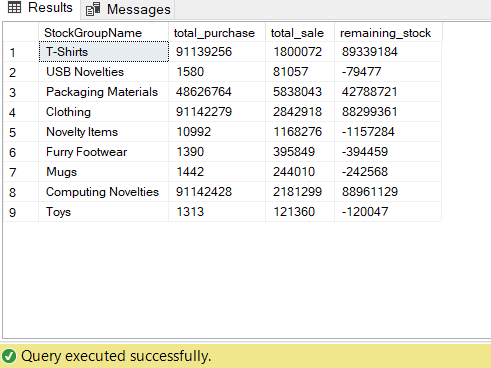
sum(ol.Quantity) as total\_sale

from Sales.OrderLines ol left join Warehouse.StockItemStockGroups ssg on ssg.StockItemID = ol.StockItemID

join Warehouse.StockGroups sg on sg.StockGroupID = ssg.StockGroupID

group by sg.StockGroupName

) s on p.StockGroupName= s.StockGroupName



1. List of Cities in the US and the stock item that the city got the most deliveries in 2016. If the city did not purchase any stock items in 2016, print “No Sales”.

with cte as (

select temp2.CityID, temp2.CityName,temp2.StockItemID, temp2.StockItemName,temp2.received\_quantity,ctr.CountryName

from

(

select temp.CityID, temp.CityName,temp.StockItemID, temp.StockItemName,temp.received\_quantity

from

(

select ct.CityID, ct.CityName,ol.StockItemID, i.StockItemName,

sum(ol.Quantity) as received\_quantity,

ROW\_NUMBER() over (partition by ct.CityName order by sum(ol.Quantity) desc) as ranking

from Sales.OrderLines ol

left join Warehouse.StockItems i on ol.StockItemID = i.StockItemID

left join Sales.Orders o on o.OrderID = ol.OrderID

left join Sales.Customers c on o.CustomerID= c.CustomerID

left join Application.Cities ct on c.DeliveryCityID =ct.CityID

where datepart(year,o.ExpectedDeliveryDate) = 2016

group by ct.CityID, ct.CityName, ol.StockItemID, i.StockItemName

) temp

where temp.ranking = 1

) temp2

left join Application.Cities c on temp2.CityID = c.CityID

left join Application.StateProvinces sp on sp.StateProvinceID = c.StateProvinceID

left join Application.Countries ctr on sp.CountryID = ctr.CountryID

where ctr.CountryName = 'United States'

)

select temp\_cities.CityName, temp\_cities.StateProvinceName, temp\_cities.CountryName,

isnull(cte.StockItemName,'No Sales') as MostDeliveriesStockItem

from

(

select c.CityID, c.CityName,sp.StateProvinceName, ctr.CountryName

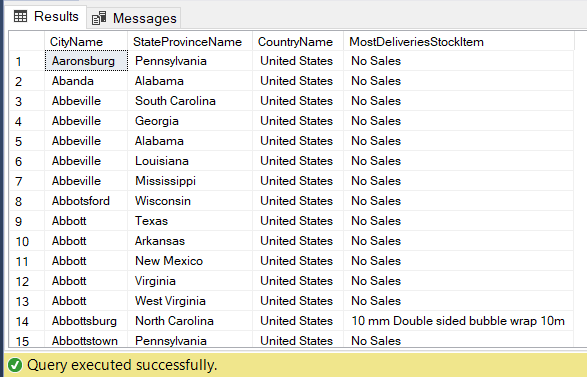
from Application.Cities c

left join Application.StateProvinces sp on sp.StateProvinceID = c.StateProvinceID

left join Application.Countries ctr on sp.CountryID = ctr.CountryID

where ctr.CountryName = 'United States'

)temp\_cities left join cte on temp\_cities.CityID = cte.CityID

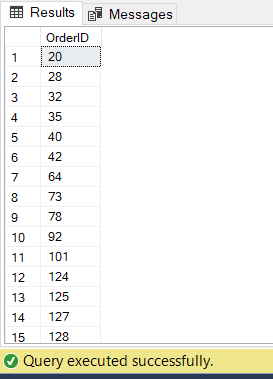


1. List any orders that had more than one delivery attempt (located in invoice table).

select distinct(inv.OrderID)

from Sales.Invoices inv

where inv.ReturnedDeliveryData like '%Receiver not present%'

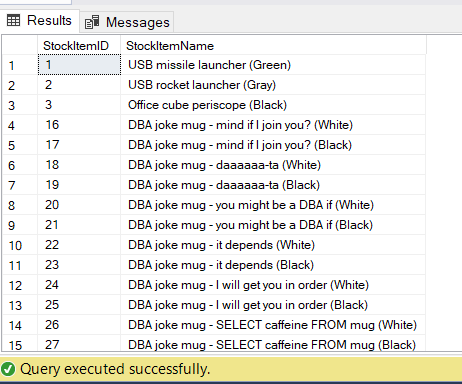


1. List all stock items that are manufactured in China. (Country of Manufacture)

select distinct si.StockItemID, si.StockItemName

from warehouse.StockItems for system\_time all si

where si.CustomFields like '%China%'



1. Total quantity of stock items sold in 2015, group by country of manufacturing.

select sti.manufacturing\_country,

sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol left join

(

select \*,

json\_value(si.CustomFields,'$.CountryOfManufacture')

as manufacturing\_country

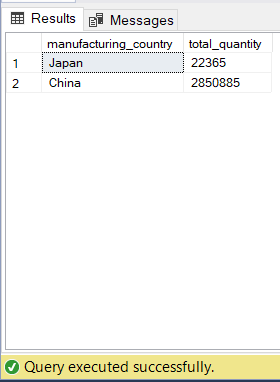
from Warehouse.StockItems si

) sti on ol.StockItemID = sti.StockItemID

left join Sales.Orders o on o.OrderID = ol.OrderID

where year(o.OrderDate) = 2015

group by sti.manufacturing\_country



1. Create a view that shows the total quantity of stock items of each stock group sold (in orders) by year 2013-2017. [Stock Group Name, 2013, 2014, 2015, 2016, 2017]

create view stockgoups

as

select StockGroupName,[2013],[2014],[2015],[2016],[2017]

from

(

select sg.StockGroupName as StockGroupName, year(o.OrderDate) as years, sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol join sales.Orders o on ol.OrderID = o.OrderID

join Warehouse.StockItems i on ol.StockItemID = i.StockItemID

join Warehouse.StockItemStockGroups ssg on ssg.StockItemID = ol.StockItemID

join Warehouse.StockGroups sg on sg.StockGroupID = ssg.StockGroupID

where year(o.OrderDate) in ( 2013, 2014, 2015, 2016, 2017)

group by sg.StockGroupName, year(o.OrderDate)

) p

pivot

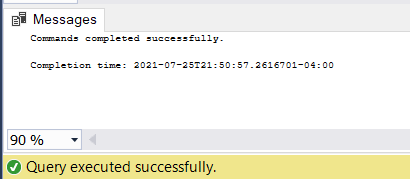
(

sum(p.total\_quantity)

for years in

([2013],[2014],[2015],[2016],[2017])

)as pivotTable



1. Create a view that shows the total quantity of stock items of each stock group sold (in orders) by year 2013-2017. [Year, Stock Group Name1, Stock Group Name2, Stock Group Name3, … , Stock Group Name10]

select years, [Clothing], [USB Novelties], [Computing Noveilties],[Airline Novelties], [Novelty Items],[T-Shirts], [Mugs],[Furry Footwear],[Toys], [Packaging Materials]

from

(

select year(o.OrderDate) as years, sg.StockGroupName as StockGroupName, sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol join sales.Orders o on ol.OrderID = o.OrderID

join Warehouse.StockItems i on ol.StockItemID = i.StockItemID

join Warehouse.StockItemStockGroups ssg on ssg.StockItemID = ol.StockItemID

join Warehouse.StockGroups sg on sg.StockGroupID = ssg.StockGroupID

where year(o.OrderDate) in ( 2013, 2014, 2015, 2016, 2017)

group by sg.StockGroupName, year(o.OrderDate)

) p

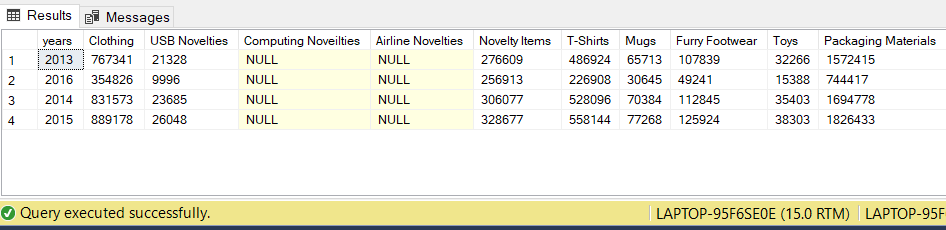
pivot

(

sum(p.total\_quantity)

for StockGroupName in ([Clothing], [USB Novelties], [Computing Noveilties],[Airline Novelties], [Novelty Items],[T-Shirts], [Mugs],[Furry Footwear],[Toys], [Packaging Materials])

) as pivotTable2



1. Create a function, input: order id; return: total of that order. List invoices and use that function to attach the order total to the other fields of invoices.

create function invoicetotal(@invoiceId int)

returns table

as

return

(

select p.total\_quantity

from

(

select i.InvoiceID, sum(il.Quantity \* il.UnitPrice) as total\_quantity

from Sales.Invoices i left join Sales.InvoiceLines il on il.InvoiceID = i.InvoiceID

--input is invoice id so use left join to include all invoice id

where i.InvoiceID = @invoiceId

group by i.InvoiceID

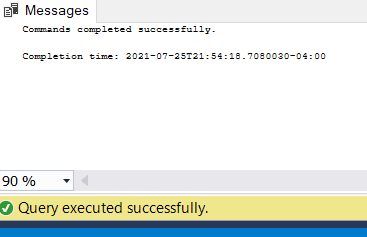
) p

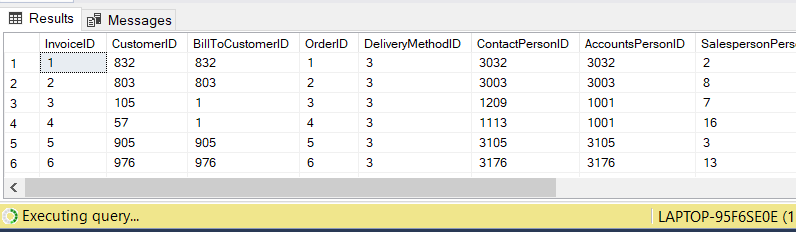
);

select \* ,

( select \* from invoicetotal(i.InvoiceID)) order\_total

from Sales.Invoices i





1. Create a new table called ods.Orders. Create a stored procedure, with proper error handling and transactions, that input is a date; when executed, it would find orders of that day, calculate order total, and save the information (order id, order date, order total, customer id) into the new table. If a given date is already existing in the new table, throw an error and roll back. Execute the stored procedure 5 times using different dates.

create table Orders2

(

orderdate date not null,

orderid int ,

customerid int ,

totalquantity int

);

create procedure dateorder

@orderdate date

as

begin try

SET NOCOUNT ON

begin tran

insert into Orders2

select o.OrderDate, o.OrderID, o.CustomerID,sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol join Sales.Orders o on ol.OrderID = o.OrderID

where o.OrderDate = @orderdate

group by o.OrderID, o.CustomerID,o.OrderDate

commit tran

end try

begin catch

SELECT ERROR\_MESSAGE() AS [Error Message]

if @orderdate in (select orderdate from Orders2)

rollback tran;

else

commit tran;

end catch

--try 2016-03-03 2016-03-04 2016-03-05 2016-03-07

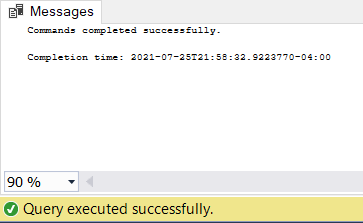
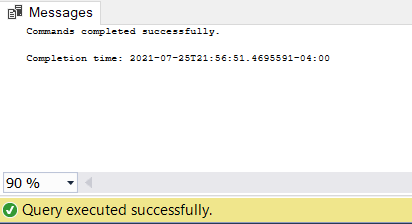
exec dateorder @orderdate = '2013-01-01'

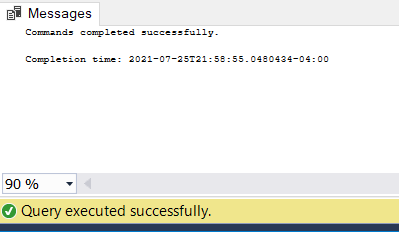
exec dateorder @orderdate = '2016-03-03'

exec dateorder @orderdate = '2016-03-05'

exec dateorder @orderdate = '2016-03-07'

exec dateorder @orderdate = '2016-03-05'





1. Create a new table called ods.StockItem. It has following columns: [StockItemID], [StockItemName] ,[SupplierID] ,[ColorID] ,[UnitPackageID] ,[OuterPackageID] ,[Brand] ,[Size] ,[LeadTimeDays] ,[QuantityPerOuter] ,[IsChillerStock] ,[Barcode] ,[TaxRate] ,[UnitPrice],[RecommendedRetailPrice] ,[TypicalWeightPerUnit] ,[MarketingComments] ,[InternalComments], [CountryOfManufacture], [Range], [Shelflife]. Migrate all the data in the original stock item table.

select StockItemID,

StockItemName,

SupplierID,

ColorID,

UnitPackageID,

OuterPackageID,

Brand,

Size,

LeadTimeDays,

QuantityPerOuter,

IsChillerStock,

Barcode,

TaxRate,

UnitPrice,

RecommendedRetailPrice,

TypicalWeightPerUnit,

MarketingComments,

InternalComments,

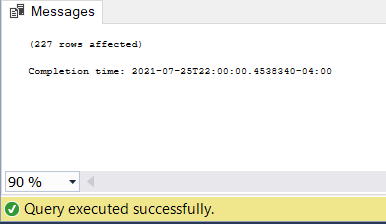
json\_value(CustomFields,'$.CountryOfManufacture') as CountryOfManufacture,

json\_value(CustomFields,'$.Range') as Range,

json\_value(CustomFields,'$.ShelfLife') as Shelflife

into odsStockItem

from Warehouse.StockItems



1. Rewrite your stored procedure in (21). Now with a given date, it should wipe out all the order data prior to the input date and load the order data that was placed in the next 7 days following the input date.

create table odsOrders

(

orderdate date not null,

orderid int ,

customerid int ,

totalquantity int

);

create procedure dateorder

@orderdate date

as

begin try

SET NOCOUNT ON

begin tran

insert into odsOrders

select o.OrderDate, o.OrderID, o.CustomerID,sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol join Sales.Orders o on ol.OrderID = o.OrderID

where o.OrderDate between @orderdate and dateadd(dd,7,@orderdate)

group by o.OrderID, o.CustomerID,o.OrderDate

except

select o.OrderDate, o.OrderID, o.CustomerID,sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol join Sales.Orders o on ol.OrderID = o.OrderID

where o.OrderDate between @orderdate and dateadd(dd,-7,@orderdate)

group by o.OrderID, o.CustomerID,o.OrderDate

commit tran

end try

begin catch

SELECT ERROR\_MESSAGE() AS [Error Message]

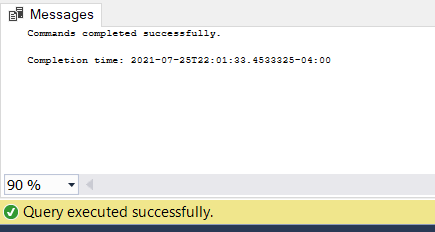
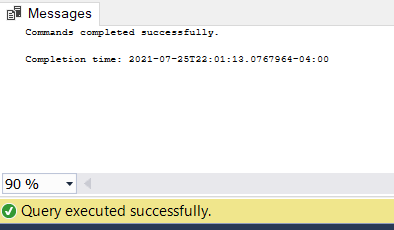
if @orderdate in (select orderdate from odsOrders)

rollback tran;

else

commit tran;

end catch



1. Consider the JSON file:

{

"PurchaseOrders":[

{

"StockItemName":"Panzer Video Game",

"Supplier":"7",

"UnitPackageId":"1",

"OuterPackageId":[

6,

7

],

"Brand":"EA Sports",

"LeadTimeDays":"5",

"QuantityPerOuter":"1",

"TaxRate":"6",

"UnitPrice":"59.99",

"RecommendedRetailPrice":"69.99",

"TypicalWeightPerUnit":"0.5",

"CountryOfManufacture":"Canada",

"Range":"Adult",

"OrderDate":"2018-01-01",

"DeliveryMethod":"Post",

"ExpectedDeliveryDate":"2018-02-02",

"SupplierReference":"WWI2308"

},

{

"StockItemName":"Panzer Video Game",

"Supplier":"5",

"UnitPackageId":"1",

"OuterPackageId":"7",

"Brand":"EA Sports",

"LeadTimeDays":"5",

"QuantityPerOuter":"1",

"TaxRate":"6",

"UnitPrice":"59.99",

"RecommendedRetailPrice":"69.99",

"TypicalWeightPerUnit":"0.5",

"CountryOfManufacture":"Canada",

"Range":"Adult",

"OrderDate":"2018-01-025",

"DeliveryMethod":"Post",

"ExpectedDeliveryDate":"2018-02-02",

"SupplierReference":"269622390"

}

]

}

Looks like that it is our missed purchase orders. Migrate these data into Stock Item, Purchase Order and Purchase Order Lines tables. Of course, save the script.

declare @json nvarchar(max) = N'[

{

"PurchaseOrders":[

{

"StockItemName":"Panzer Video Game",

"Supplier":"7",

"UnitPackageId":"1",

"OuterPackageId":[

6,

7

],

"Brand":"EA Sports",

"LeadTimeDays":"5",

"QuantityPerOuter":"1",

"TaxRate":"6",

"UnitPrice":"59.99",

"RecommendedRetailPrice":"69.99",

"TypicalWeightPerUnit":"0.5",

"CountryOfManufacture":"Canada",

"Range":"Adult",

"OrderDate":"2018-01-01",

"DeliveryMethod":"Post",

"ExpectedDeliveryDate":"2018-02-02",

"SupplierReference":"WWI2308"

},

{

"StockItemName":"Panzer Video Game",

"Supplier":"5",

"UnitPackageId":"1",

"OuterPackageId":"7",

"Brand":"EA Sports",

"LeadTimeDays":"5",

"QuantityPerOuter":"1",

"TaxRate":"6",

"UnitPrice":"59.99",

"RecommendedRetailPrice":"69.99",

"TypicalWeightPerUnit":"0.5",

"CountryOfManufacture":"Canada",

"Range":"Adult",

"OrderDate":"2018-01-025",

"DeliveryMethod":"Post",

"ExpectedDeliveryDate":"2018-02-02",

"SupplierReference":"269622390"

}

]

}

]'

select \*

from openjson(@json)

with(

StockItemName VARCHAR '$.StockItemName',

SupplierID int '$.Supplier',

UnitPackageID int '$.UnitPackageId',

OuterPackageId int '$.OuterPackageId',

Brand nvarchar '$.Brand',

LeadTimeDays int '$.LeadTimeDays',

QuantityPerOuter int '$.QuantityPerOuter',

TaxRate decimal '$.TaxRate',

UnitPrice decimal '$.UnitPrice',

RecommendedRetailPrice decimal '$.RecommendedRetailPrice',

TypicalWeightPerUnit decimal '$.TypicalWeightPerUnit',

[CustomFields] nvarchar '$.CountryOfManufacture',

OrderDate date '$.OrderDate',

DeliveryMethod nvarchar '$.DeliveryMethod',

ExpectedDeliveryDate date '$.ExpectedDeliveryDate',

SupplierReference nvarchar '$.SupplierReference'

)

-- continuing...

1. Revisit your answer in (19). Convert the result in JSON string and save it to the server using TSQL FOR JSON PATH.

select years, [Clothing], [USB Novelties], [Computing Noveilties],[Airline Novelties], [Novelty Items],[T-Shirts], [Mugs],[Furry Footwear],[Toys], [Packaging Materials]

from

(

select year(o.OrderDate) as years, sg.StockGroupName as StockGroupName, sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol join sales.Orders o on ol.OrderID = o.OrderID

join Warehouse.StockItems i on ol.StockItemID = i.StockItemID

join Warehouse.StockItemStockGroups ssg on ssg.StockItemID = ol.StockItemID

join Warehouse.StockGroups sg on sg.StockGroupID = ssg.StockGroupID

where year(o.OrderDate) in ( 2013, 2014, 2015, 2016, 2017)

group by sg.StockGroupName, year(o.OrderDate)

) p

pivot

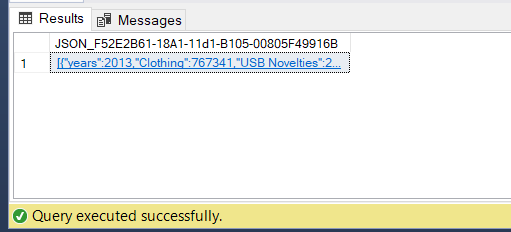
(

sum(p.total\_quantity)

for StockGroupName in ([Clothing], [USB Novelties], [Computing Noveilties],[Airline Novelties], [Novelty Items],[T-Shirts], [Mugs],[Furry Footwear],[Toys], [Packaging Materials])

) as pivotTable2

for json path



1. Revisit your answer in (19). Convert the result into an XML string and save it to the server using TSQL FOR XML PATH.

select years, [Clothing], [USB Novelties], [Computing Noveilties],[Airline Novelties], [Novelty Items],[T-Shirts], [Mugs],[Furry Footwear],[Toys], [Packaging Materials]

from

(

select year(o.OrderDate) as years, sg.StockGroupName as StockGroupName, sum(ol.Quantity) as total\_quantity

from Sales.OrderLines ol join sales.Orders o on ol.OrderID = o.OrderID

join Warehouse.StockItems i on ol.StockItemID = i.StockItemID

join Warehouse.StockItemStockGroups ssg on ssg.StockItemID = ol.StockItemID

join Warehouse.StockGroups sg on sg.StockGroupID = ssg.StockGroupID

where year(o.OrderDate) in ( 2013, 2014, 2015, 2016, 2017)

group by sg.StockGroupName, year(o.OrderDate)

) p

pivot

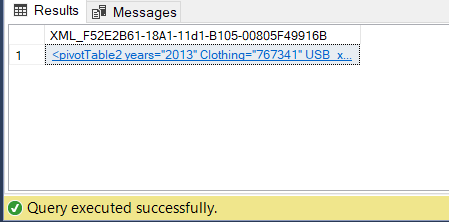
(

sum(p.total\_quantity)

for StockGroupName in ([Clothing], [USB Novelties], [Computing Noveilties],[Airline Novelties], [Novelty Items],[T-Shirts], [Mugs],[Furry Footwear],[Toys], [Packaging Materials])

) as pivotTable2

for xml auto



1. Create a new table called ods.ConfirmedDeviveryJson with 3 columns (id, date, value) . Create a stored procedure, input is a date. The logic would load invoice information (all columns) as well as invoice line information (all columns) and forge them into a JSON string and then insert into the new table just created. Then write a query to run the stored procedure for each DATE that customer id 1 got something delivered to him.

create procedure [dbo].[SaveInvoices](@date date, @cID int) as begin

insert dbo.ConfirmedDeviveryJson(value, date)

(select (Select \* from WideWorldImporters.Sales.Invoices i

join WideWorldImporters.Sales.InvoiceLines il on il.InvoiceID = i.InvoiceID

and i.InvoiceDate = @date and i.CustomerID = @cID

for json auto) as value, @date)

end;

-- adding customer dates information using a cursor

select distinct i.InvoiceDate as dates into #dates

from WideWorldImporters.Sales.Invoices i

where i.CustomerID = 1

declare cur cursor for select dates from #dates

declare @d date

open cur fetch next from cur into @d

while @@FETCH\_STATUS = 0

begin

exec dbo.SaveInvoices @d

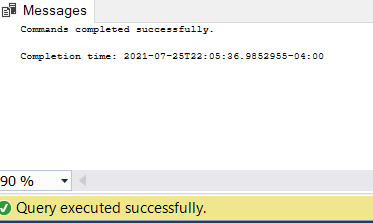
fetch next from cur into @d

end;

close cur

deallocate cur

drop table #dates



**28.Write a short essay (400-750 words) talking about your understanding of transactions, locks and isolation levels.**

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must have four properties: ACID. Atomicity means either all of the data modifications are performed, or none of them are performed. Consistency means when completed, a transaction must leave all data in a consistent state. Isolation means modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. Durability means after a fully durable transaction has completed, its effects are permanently in place in the system.

Usually, there are 2 ways to manage transactions, pessimistic concurrency control and optimistic concurrency control. Pessimistic concurrency control uses locks. Because of this, you have a lower risk of having a rolling back, if something is locked and you want to update, all you need to do is to wait for locks to be released. This is more commonly used in industry, because most systems want data accuracy and few rolling back possible. It’s ok to wait than redo analysis for people doing business.

Optimistic concurrency control doesn't use locks at all, it uses system row versioning or time stamp, if 2 transactions are trying to update the same resources, whoever with a command first goes first. So it has a high risk of rolling back transactions but low on waiting times. There are 2 ways in sql server for optimistic concurrency control. One is to use In memory oltp table and the other is to use a snapshot.

In memory oltp table uses a time stampe. When you are trying to commit some changes to it, it will compare the timestamp when you get the data and the timestamp of that row's last updated time and tell you if your changes are allowed or not. In memory oltp is special because we will have a copy on the hard drive, but only for system re-read data and put them into the ram. In memory oltp table are in the ram, so it’s fast.

Snapshot is close to the oltp table. The way it works as: whenever a transaction requires a copy of data which it can work on, the system will create a snapshot of current table/data,a snapshot means a duplicated copy, then system will send this duplicated copy to that transaction, meaning the transaction will work with this duplicated copied table on hard drive. Whenever you are ready to finish the transaction and do some changes to data, it’s going to use a system row versioning for that, which basically compares versions of rows data to see changes you made are allowed.

Pessimistic concurrency control uses locks to manage transactions. To category locks by use, we have shared lock, update lock and exclusive lock.

shared lock works as: when a transaction is trying to read data, apply shared lock first to the resource. Somehow, if there is another lock which is also a shared lock, you can also read it. So shared lock means multiple transactions can read the source at the same time.

update lock is not shared. When you are trying to update sth, then you first change shared lock into an update lock which tells others you are trying to update some data in this dataset. If others try to read this data, in which they can’t apply their shared lock and read while you are trying to update things. So, when your transaction is trying to update things nobody can read until you finish.

exclusive lock happens when the actual updating starts. It is exclusive because once you change a lock into an exclusive lock , it’s no longer coming back. The exclusive lock will stay there forever until your current transaction finishes, which tells others you have updated the data.

SQL Server isolation levels are used to define the degree to which one transaction must be isolated from resource or data modifications made by other concurrent transactions. From the lowest to the highest:

1.read uncommitted: is the lowest isolation level, basically telling the system you are fine with reading uncommitted changes, which means you don’t care about any locks and tell the system to give you whatever it has at this moment. Read uncommitted just reads anything available on the hard drive at this moment.

2.read committed: is a system default isolation level, which means when you begin a transaction, you actually use read committed isolation level. Basically this is like how locks work. Read committed is when it tries to read, it will check locks, if there is any lock other than shared lock, it will wait.

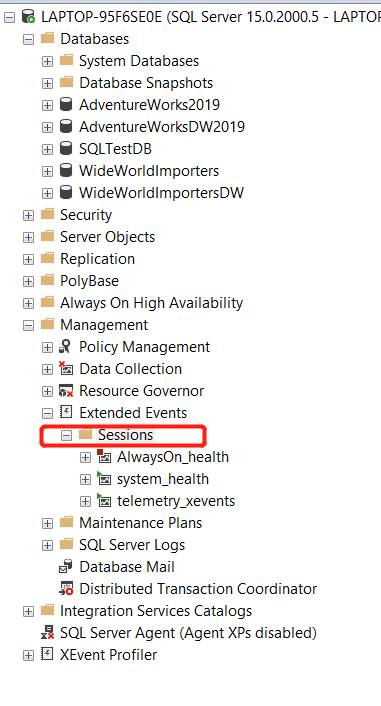
3.repeatable read: means you are not releasing your shared lock even after you finish your reading. You are keeping your shared lock and nobody else can change the data, ih which way help you make sure data is consistent before your transaction is finished.

4.serializable: apply locks on a wider range than repeatable reads which apply locks on every row. It won’t allow any inserting into that range until your transaction is finished.

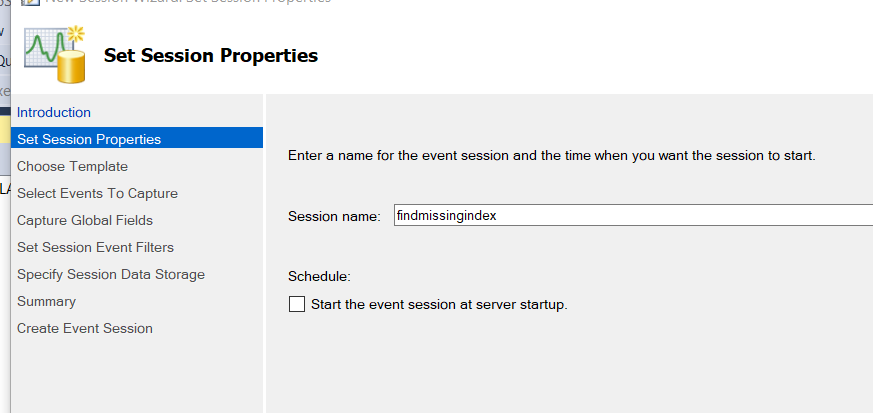
**29.Write a short essay (300+ words, plus screenshots) talking about a scenario: It is Monday morning and your boss just told you that “he got feedback from a random average user that the ‘website’ was very slow and even threw db related errors during the weekend”. What should you do?**

First, I will reply to my manager about the major steps, the methods I will try and the time estimated to fix this performance problem. And also I might ask some suggestions from my manager if possible.

To locate the problem, the first thing I will look into is always performance monitoring, which aims to find out where the bottleneck is.The most commonly used methods are execution plan, tuning advisor, extended events, DMV, system stored procedures, or DBCC trace flags.



For example, if looking for a missing index. I can perform extended events and set a session from clicking 1.Management-> Extended Events -> Session and click on the New Session Wizard.

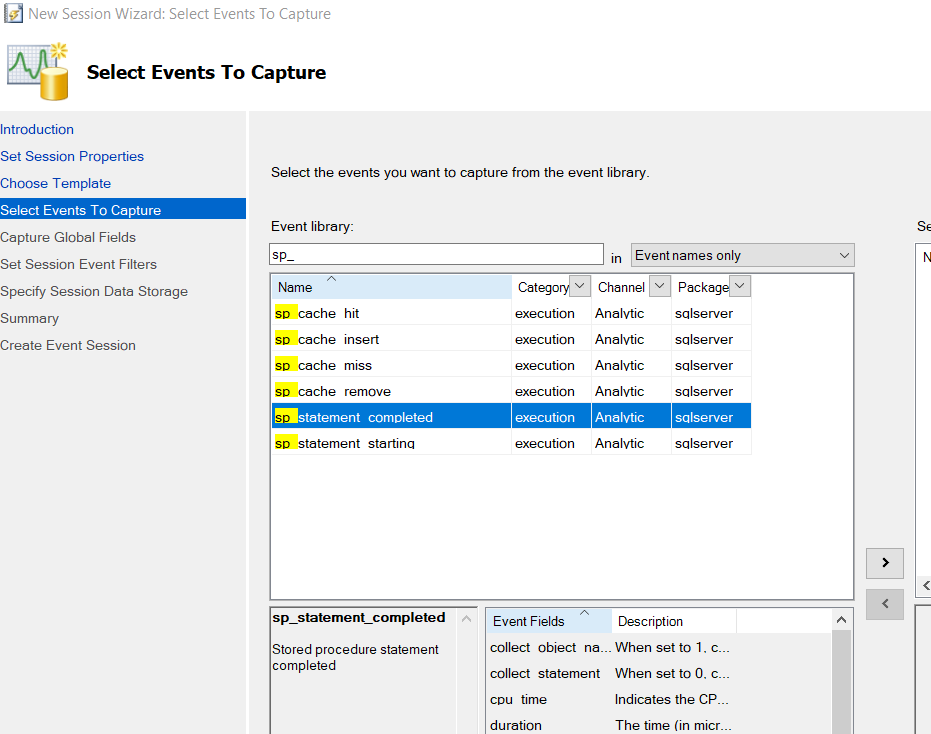
2.Then set session propoerties by naming a name of this event :

3. Then at the event library select the event to capture, usually we can select:

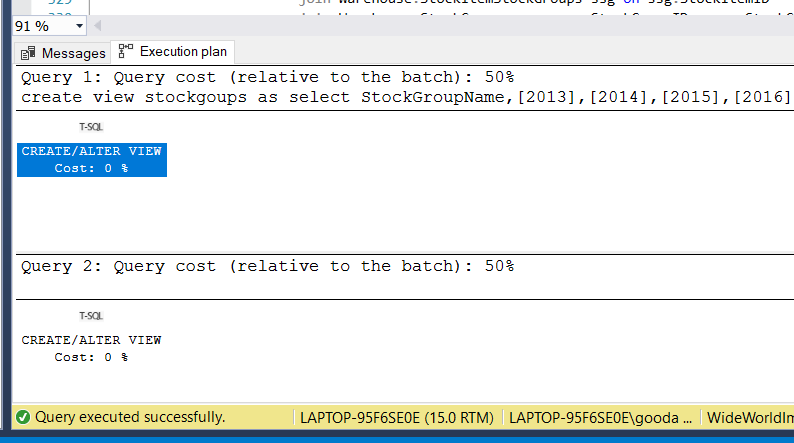
sql\_statement\_completed,

sp\_statement\_completed

sql\_batch\_completed



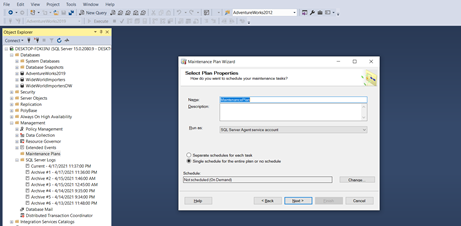
After setting up all properties, we can find the event log on session by clicking the name of the session.There are also alternative ways to do performance tuning, for example, we can look at execution plan for tables to find if there is a table scan:

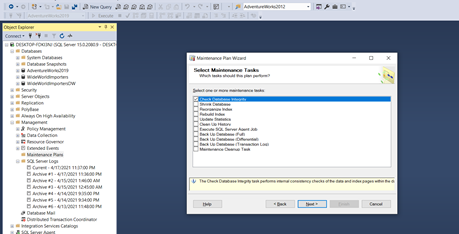


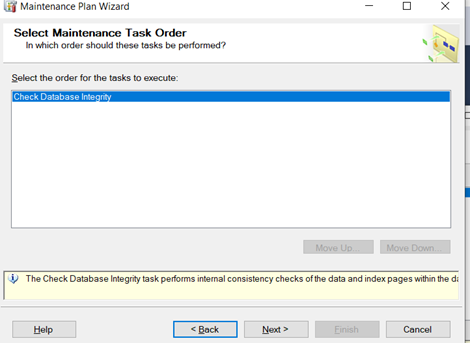
Aftering trying the above methods, I can find out why the server services are slow and do corresponding actions against it, for example creating query hints if needed, mostly NOLOCK or creating indexes. And also change the way the queries are written if needed. Normally, we don’t want unnecessary groupby, too many correlated subqueries. We want to remove conditions on where clauses to a join condition if needed. We want to find missing indexes but don’t want to try too much to become an overhead. Also be aware of using bulk insert instead of inserting individually.

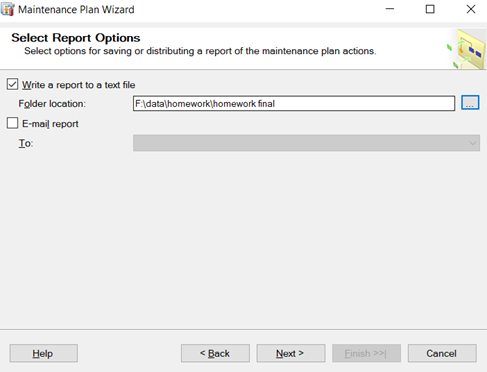
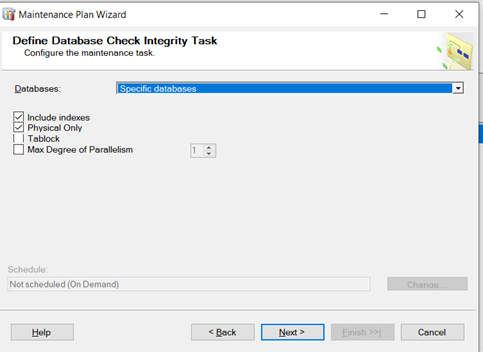
Next is to compare the original performance with new performance from the execution plan or other sets of extended events and find out if it is actually improved.

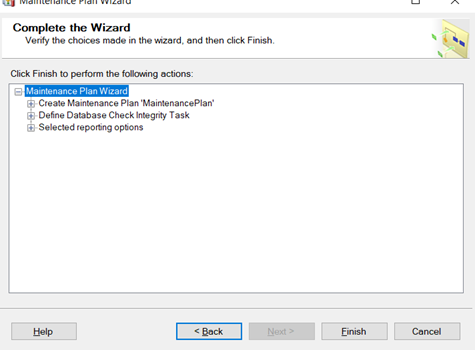
**30.Create a maintenance job. Create extended events monitoring deadlocks and long-run queries. Use a step by step screenshot for this assignment.**

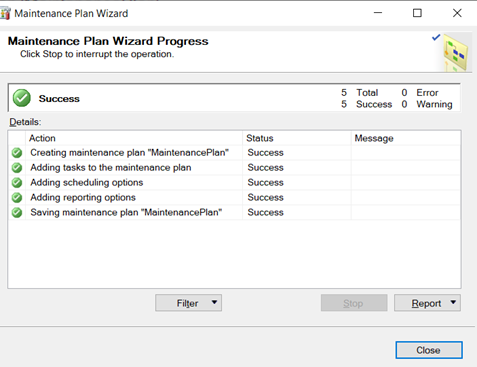






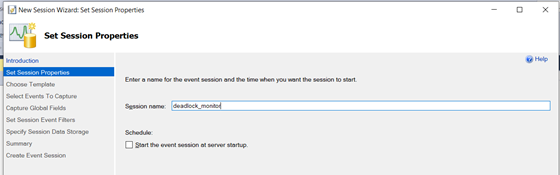




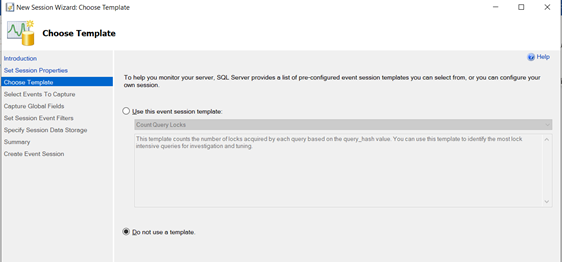


2:extended events monitoring deadlocks

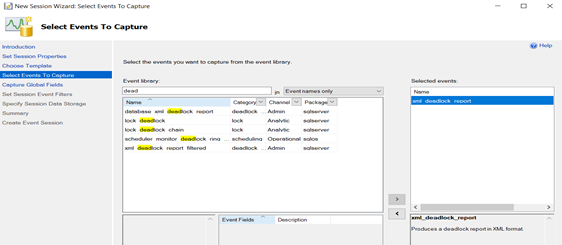
step1



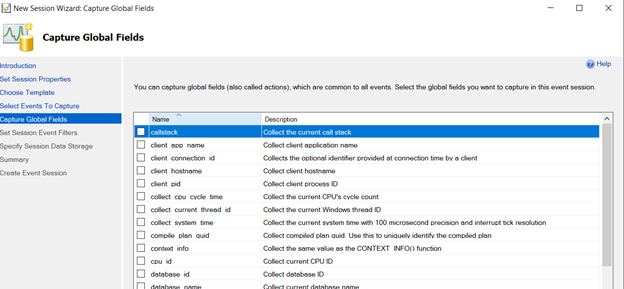
Step2



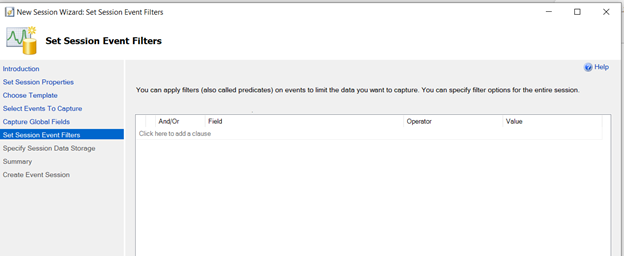
Step3



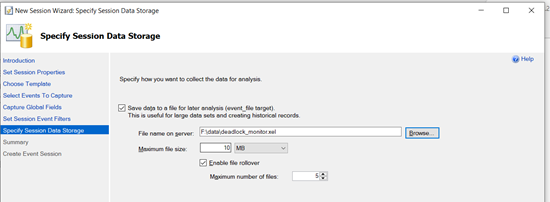
Step 4



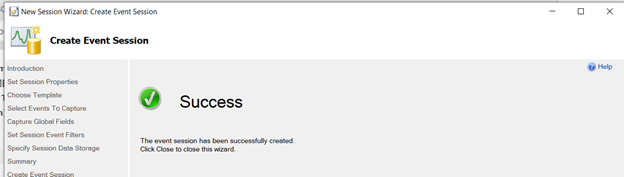
Step5



Step 6



Step7



Step8 start event

Assignments 31 - 34 are group assignments.

**31.Write a short essay (500+ words) talking about a scenario:**

**Good news everyone! We (Wide World Importers) just brought out a small company called “Adventure works”! Now that bike shop is our sub-company. The first thing of all works pending would be to merge the user logon information, person information (including emails, phone numbers) and products (of course, add category, colors) to WWI database. Include screenshot, mapping and query.**

**1. Introduction**

Based on the scenario from Wide World Importer, we need to merge relevant information from it’s newly acquired company “Adventure Works”(AW) into the “Wide World Importers”(WWI) database.

1). Person Table

In order to find combined information of user logon, persons and products, we need to map relevant columns in both datasets. First, we analyzed the WWI database. We found the basic people’s information all included in the application.people table. It includes information like people’s full name, email address, phone number, login information and Personal Identity. WWI As the main company, we need to use its database as the main database, and merge the acquired company database according to the main database template.

However, we compare people’s information in the AW database. The relationship between person and other tables is shown below:

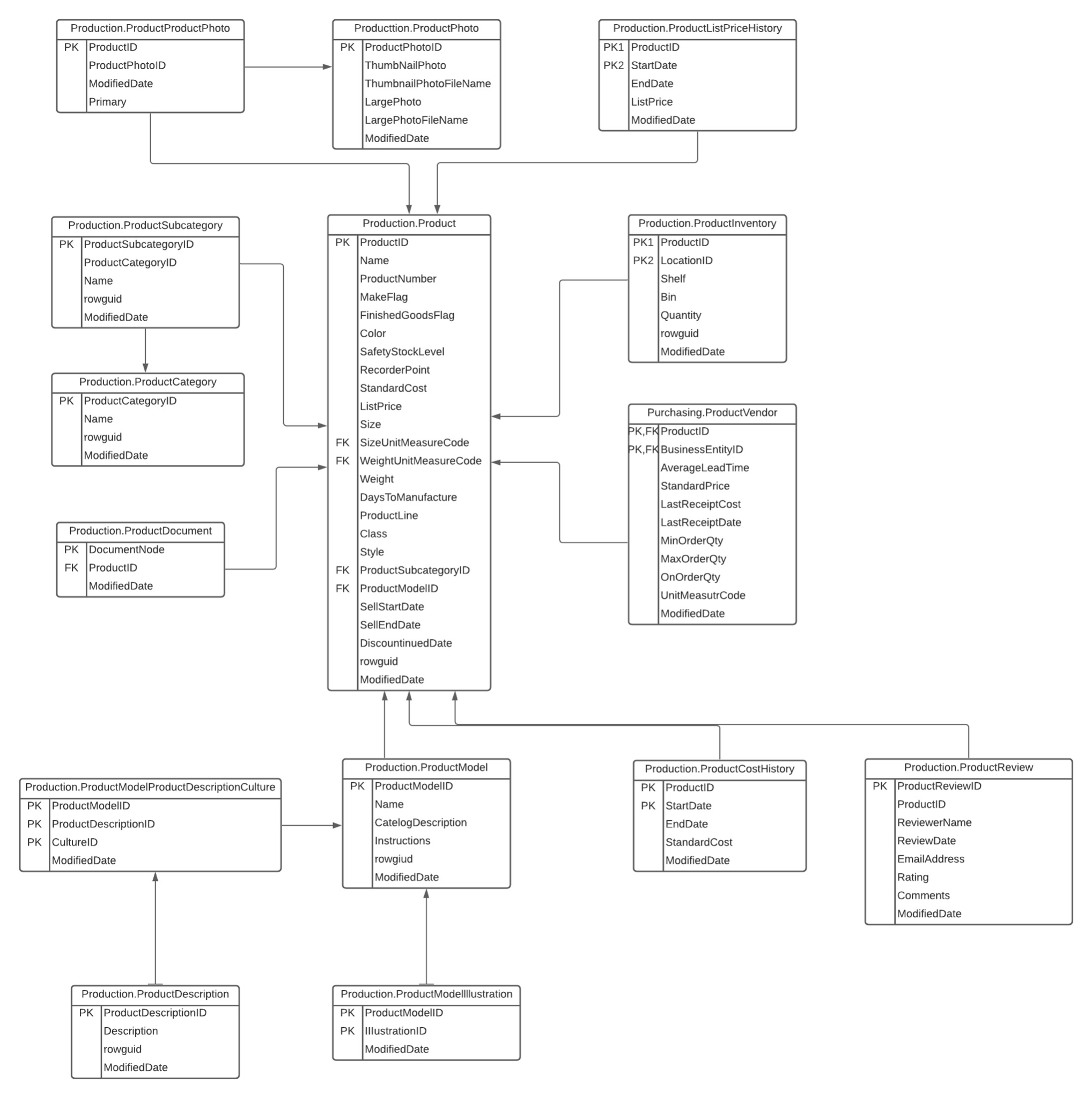
Diagram

Description automatically generated

As can be seen from the above ERD of AW, we can get people’s First Name, people’s MiddleName, people’s lastname ,people’s EmailAddress, people’s phonenumber and account’s password. Other information like city’s information we can just use WWI’s database.

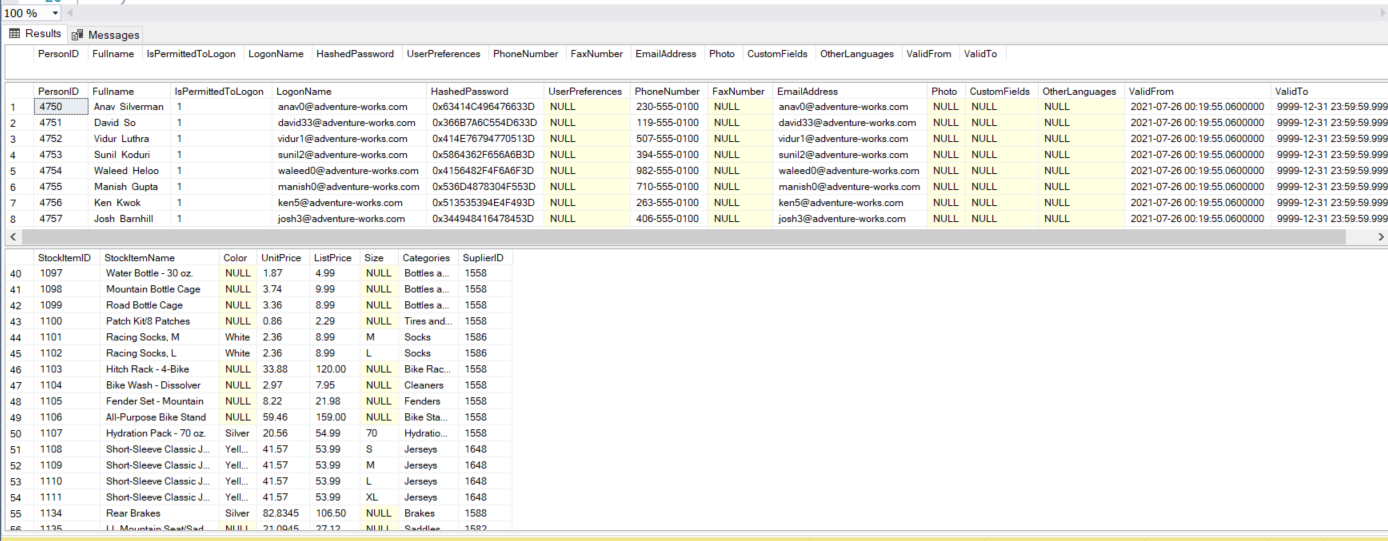
Firstly, we Declared a variable, @maxpersonid to store the maximum PersonID and add the maximum value to BusinessEntityID to make sure the data integrity of primary key constraint. And we will use AW’S passwordHash as hashpassword in WWI and after we analyzed the data stored in the WWI database logonname, we thought we could use the email address in the AW personal information as logonname. And we combine first name, middle name and last name in Adventure Works database to match with Application.People. Besides, we set the current system time as the valid start time and using 12/31/9999 23:59:59.9999 as valid end time as WWI do. Leave photo, customerfield, UserPreferences, fax number, OtherLanguages blank to be filled in later.

2) Product Table



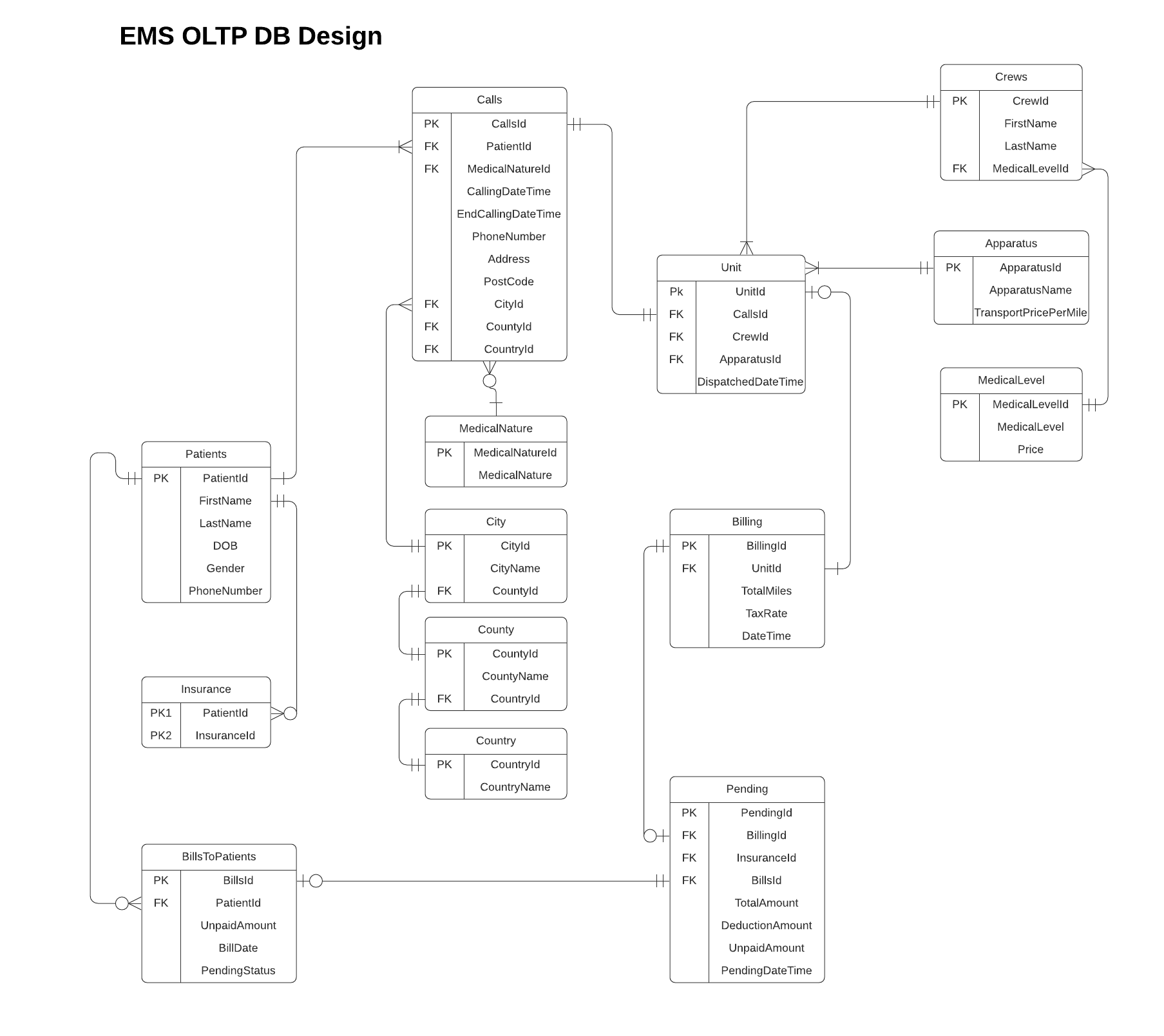
As can be seen from the above ERD, we can get product name, product color, list price, size, categories from AW database. And we can merge this information into the WWI database warehouse.stockitems table. Also, =we Declared a variable, @maxid to store the maximum ProductID and add the maximum value to StockItemID to make sure the data integrity of primary key constraint. And we choose product SubcategoryID data as Categories data in WWI database instead of CategoryID since SubcategoryID is a group of parts and CategoryID is a group of bike type. And for supplier part, in AW purchasing.productvendor we can find the detail of supplier, so we generate the information. We set AW purchasing.productvendor’s businessEntityID as suplierID in WWI.

3. Result



**32.Database Design: OLTP db design request for EMS business:**

1. **when people call 911 for medical emergency, 911 will dispatch UNITs to the given address.**
2. **A UNIT means a crew on an apparatus (Fire Engine, Ambulance, Medic Ambulance, Helicopter, EMS supervisor).**
3. **A crew member would have a medical level (EMR, EMT, A-EMT, Medic).**
4. **All the treatments provided on scene are free. If the patient needs to be transported, that’s where the bill comes in.**
5. **A bill consists of Units dispatched (Fire Engine and EMS Supervisor are free), crew members provided care (EMRs and EMTs are free), Transported miles from the scene to the hospital (Helicopters have a much higher rate, as you can image) and tax (Tax rate is 6%).**
6. **Bill should be sent to the patient insurance company first. If there is a deductible, we send the unpaid bill to the patient only.**
7. **Don’t forget about patient information, medical nature and bill paying status.**



**34.Study the Wide World Importers DW. Think the integration schema is the ODS. Come up with a TSQL Stored Procedure driven solution to move the data from WWI database to ODS, and then from the ODS to the fact tables and dimension tables. By the way, WWI DW is a galaxy schema db. Requirements:**

1. **Luckly, we only start with 1 fact: Order. Other facts can be ignored for now.**
2. **Add a new dimension: Country of Manufacture. It should be given on top of Stock Items.**
3. **Write script(s) and stored procedure(s) for the entire ETL from WWI db to DW.**

use WideWorldImportersDW

select \* from Integration.Order\_Staging

select \* from Integration.Lineage

use WideWorldImporters

select \* from Sales.OrderLines

select \* from Sales.Orders

select \* from Sales.Customers

select \* from Warehouse.StockItems

select \* from Sales.Invoices

select \* from Sales.InvoiceLines

select c.DeliveryCityID as City, c.CustomerID as Customer, si.StockItemID as StockItem, o.OrderDate as OrderDate, o.SalespersonPersonID as Salesperson,

sic.PackedByPersonID as Picker, o.BackorderOrderID as BackorderID, ol.Description as Description, pt.PackageTypeName as Package,

ol.Quantity AS Quantity, ol.UnitPrice as UnitPrice, ol.TaxRate as TaxRate, ol.Quantity\*ol.UnitPrice as TotalExcludingTax,

ol.Quantity\*ol.UnitPrice\*ol.TaxRate\*0.01 as TaxAmount, ol.Quantity\*ol.UnitPrice\*(1+ol.TaxRate)\*0.01 as TotalIncludingTax

into #temp

from Sales.OrderLines ol join Sales.Orders o on ol.OrderId = o.OrderId

join Sales.Customers c on c.CustomerID = o.CustomerID

join Warehouse.StockItems si on si.StockItemID = ol.StockItemID

join Sales.Invoices sic on sic.OrderID = o.OrderID

join Warehouse.PackageTypes pt on pt.PackageTypeID = ol.PackageTypeID

--continuing