SQL Assignments

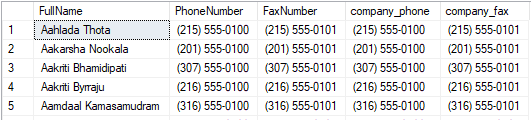
SQL related assignments will be on Wide World Importers Database if not otherwise introduced.

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).

select distinct a.FullName, a.PhoneNumber, a.FaxNumber, b.PhoneNumber as company\_phone, b.FaxNumber as company\_fax

from Application.People a

join Sales.Customers b on a.PersonID = b.PrimaryContactPersonID



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

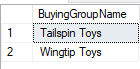
select distinct c.BuyingGroupName

from Application.People a

join Sales.Customers b on a.PersonID = b.PrimaryContactPersonID

join Sales.BuyingGroups c on b.BuyingGroupID = c.BuyingGroupID

where a.PhoneNumber = b.PhoneNumber



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

select CustomerID

from sales.Orders

except

select CustomerID

from Sales.Orders

where OrderDate > '2016-01-01'

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

select a.StockItemName ,b.QuantityOnHand

from Warehouse.StockItems a

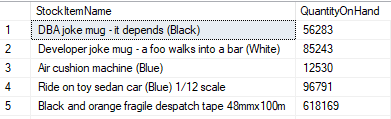
join Warehouse.StockItemHoldings b on a.StockItemID = b.StockItemID

where a.StockItemID in (select aa.StockItemID

from Purchasing.PurchaseOrderLines aa

join Purchasing.PurchaseOrders bb on aa.PurchaseOrderID = bb.PurchaseOrderID

where year(bb.OrderDate) = '2013')

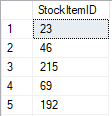


1. List of stock items that have at least 10 characters in description.

select distinct StockItemID

from Purchasing.PurchaseOrderLines

where len(Description) >= 10



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

select e.StockItemID

from Sales.Customers a

join Application.Cities b on a.PostalCityID = b.CityID

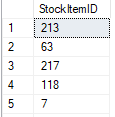
join Application.StateProvinces c on b.StateProvinceID = c.StateProvinceID

join Sales.Orders d on a.CustomerID = d.CustomerID

join Warehouse.StockItemTransactions e on a.CustomerID = e.CustomerID

where c.StateProvinceName not in ('Alabama', 'Georgia')

and d.OrderDate = '2014'



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

select b.StateProvinceName, avg(datediff(DAY, e.OrderDate, cast(d.ConfirmedDeliveryTime as date))) as Avg\_Processing\_Time

from Application.Cities a

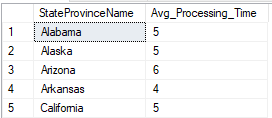
join Application.StateProvinces b on a.StateProvinceID = b.StateProvinceID

join Sales.Customers c on c.PostalCityID = a.CityID

join Sales.Invoices d on c.CustomerID = d.CustomerID

join Sales.Orders e on d.OrderID = e.OrderID

group by b.StateProvinceName



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

select b.StateProvinceName, month(e.OrderDate) as Month, avg(datediff(DAY, e.OrderDate, cast(d.ConfirmedDeliveryTime as date))) as Avg\_Processing\_Time

from Application.Cities a

join Application.StateProvinces b on a.StateProvinceID = b.StateProvinceID

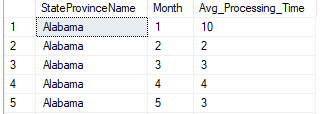
join Sales.Customers c on c.PostalCityID = a.CityID

join Sales.Invoices d on c.CustomerID = d.CustomerID

join Sales.Orders e on d.OrderID = e.OrderID

group by month(e.OrderDate), b.StateProvinceName

order by 1,2



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

select aa.StockItemID

from Warehouse.StockItemHoldings as aa

join(select a.StockItemID, sum(c.OrderedOuters) as quantity

from sales.OrderLines a

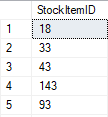
join Sales.Orders b on a.OrderID = b.OrderID

join Purchasing.PurchaseOrderLines c on c.StockItemID = a.StockItemID

where year(b.OrderDate) = '2015'

group by a.StockItemID) as bb on aa.StockItemID = bb.StockItemID

where aa.QuantityOnHand > bb.quantity



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.

select c.CustomerID, d.PhoneNumber, e.FullName

from Warehouse.StockItems a

join Sales.OrderLines b on a.StockItemID = b.StockItemID

join Sales.Orders c on b.OrderID = c.OrderID

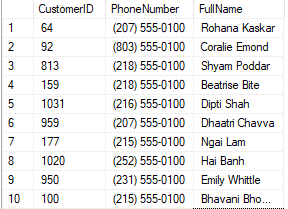
join Sales.Customers d on c.CustomerID = d.CustomerID

join Application.People e on d.PrimaryContactPersonID = e.PersonID

where year(c.OrderDate) = '2016'

and a.StockItemName like '%mug%'

and b.Quantity <= 10



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

SELECT CityName

FROM Application.Cities

FOR SYSTEM\_TIME CONTAINED IN ('2015-01-01', '9999-12-31 23:59:59.9999999')



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 a.StockItemName, d.DeliveryAddressLine1 +d.DeliveryAddressLine2 as delivery\_address,

f.StateProvinceName, e.CityName, g.CountryName,

d.CustomerName, h.FullName, d.PhoneNumber, b.Quantity

from Warehouse.StockItems a

join Sales.OrderLines b on a.StockItemID = b.StockItemID

join Sales.Orders c on c.OrderID = b.OrderID

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

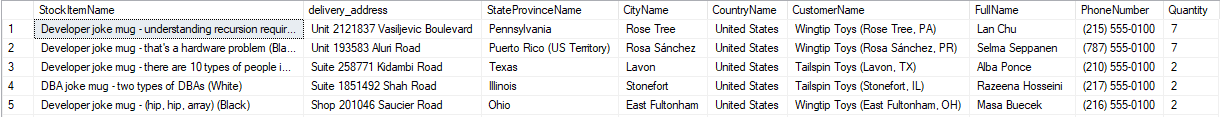
join Application.Cities e on d.DeliveryCityID = e.CityID

join Application.StateProvinces f on f.StateProvinceID = e.StateProvinceID

join Application.Countries g on g.CountryID = f.CountryID

join Application.People h on d.PrimaryContactPersonID = h.PersonID

where c.OrderDate = '2014-07-01'



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

WITH cte1 AS (

SELECT b.StockGroupID, SUM(a.OrderedOuters) AS qty\_purchased

FROM Purchasing.PurchaseOrderLines a

JOIN Warehouse.StockItemStockGroups b ON a.StockItemID = b.StockItemID

GROUP BY b.StockGroupID

),

cte2 AS (

SELECT b.StockGroupID, SUM(a.Quantity) AS qty\_sold

FROM Sales.OrderLines a

JOIN Warehouse.StockItemStockGroups b ON a.StockItemID = b.StockItemID

GROUP BY b.StockGroupID)

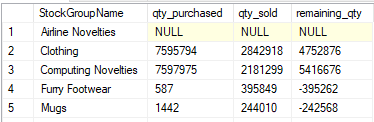
SELECT a.StockGroupName, b.qty\_purchased, c.qty\_sold,

b.qty\_purchased - c.qty\_sold AS remaining\_qty

FROM Warehouse.StockGroups a

LEFT JOIN cte1 b ON a.StockGroupID = b.StockGroupID

LEFT JOIN cte2 c ON a.StockGroupID = c.StockGroupID



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 b.StockItemID, c.DeliveryCityID, rank()over(partition by c.DeliveryCityID order by count(a.orderID) desc) rnk,

count(a.orderID) ccc

from Sales.Orders a

join Sales.OrderLines b on a.OrderID = b.OrderID

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

where year(a.OrderDate) = '2016'

group by c.DeliveryCityID,b.StockItemID

having count(a.OrderID) != 0

)

select b.CityName,

case

when c.StockItemName is not null then c.StockItemName

else 'No Sales'

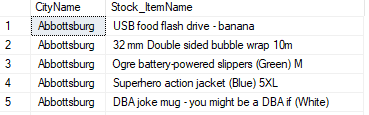
end as Stock\_ItemName

from cte a

right join Application.Cities b on a.DeliveryCityID = b.CityID

join Warehouse.StockItems c on a.StockItemID = c.StockItemID

where rnk = 1

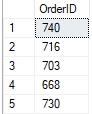


1. List any orders that had more than one delivery attempt (located in invoice table).

select OrderID

from Sales.Invoices

where ReturnedDeliveryData like '%Receiver not present%'

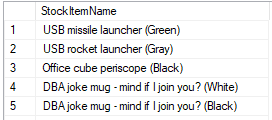


1. List all stock items that are manufactured in China. (Country of Manufacture)

select distinct StockItemName

from Warehouse.StockItems

where CustomFields like '%China%'



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

with cte as (

SELECT StockItemID,

JSON\_VALUE(CustomFields, '$.CountryOfManufacture') AS Country

FROM Warehouse.StockItems

)

select a.Country, sum(b.Quantity) qty\_sold

from cte a

join Sales.OrderLines b on a.StockItemID = b.StockItemID

join Sales.Orders c on b.OrderID = c.OrderID

where year( c.OrderDate) = 2015

group by a.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 stockitem\_qty\_byyear

AS

WITH SourceTable

AS

(SELECT SUM(a.Quantity) AS total, d.StockGroupName,

Year(b.OrderDate) AS yyear

FROM Sales.OrderLines as a

JOIN Sales.Orders AS b

ON a.OrderID = b.OrderID

JOIN Warehouse.StockItemStockGroups AS c

ON a.StockItemID = c.StockItemID

JOIN Warehouse.StockGroups AS d

ON c.StockGroupID = d.StockGroupID

GROUP BY Year(b.OrderDate), d.StockGroupName,

)

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

FROM SouceTable

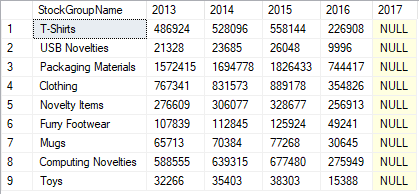
PIVOT

(MAX(total)

FOR yyear in ([2013], [2014], [2015], [2016], [2017])

) AS aa

GO



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]

CREATE VIEW stockitem\_qty\_bygroup

AS

WITH SourceTable

AS

(SELECT SUM(a.Quantity) AS total, d.StockGroupName,

Year(b.OrderDate) AS yyear

FROM Sales.OrderLines as a

JOIN Sales.Orders AS b

ON b.OrderID = a.OrderID

JOIN Warehouse.StockItemStockGroups AS c

ON a.StockItemID = c.StockItemID

JOIN Warehouse.StockGroups AS d

ON c.StockGroupID = d.StockGroupID

GROUP BY d.StockGroupName, Year(b.OrderDate)

)

SELECT yyear, Clothing, [Computing Novelties],

[Furry Footwear], Mugs, [Novelty Items], [Packaging Materials],

Toys, [T-Shirts], [USB Novelties]

FROM SourceTable

PIVOT (

MIN(total)

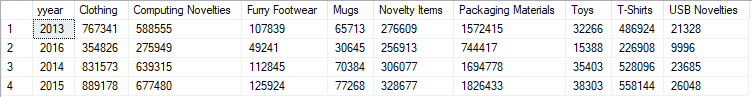
FOR StockGroupName IN (Clothing, [Computing Novelties],

[Furry Footwear], Mugs, [Novelty Items], [Packaging Materials],

Toys, [T-Shirts], [USB Novelties])) AS aa

GO

SELECT \* FROM stockitem\_qty\_bygroup



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 fun20(@OrderID INT)

RETURNS INT

BEGIN

DECLARE @TotalAmount AS DECIMAL(20,2)

SELECT @TotalAmount = SUM((a.UnitPrice \* a.Quantity))

FROM Sales.OrderLines AS a

JOIN Sales.Orders AS b

ON b.OrderID = a.OrderID

JOIN Sales.Customers AS c

ON c.CustomerID = b.CustomerID

WHERE b.OrderID = @OrderID

GROUP BY c.CustomerID, b.OrderID, b.OrderDate

RETURN @TotalAmount

END

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 ods.Orders

(CustomerID int,

OrderID int,

OrderDate Date ,

Total Decimal(20,2),

)

Go

CREATE PROC sp21\_order @OrderDate Date

AS

BEGIN

DECLARE @Exist AS INT

SET @Exist = (SELECT Count(\*) from ods.Orders

WHERE OrderDate = @OrderDate) --check if @orderdate arleady existed in the target table

BEGIN TRY

BEGIN TRAN

INSERT INTO ods.Orders(CustomerID, OrderID, OrderDate, Total)

SELECT c.CustomerID, b.OrderID, b.OrderDate,

SUM((a.UnitPrice \* a.Quantity)) AS Total

FROM Sales.OrderLines AS a

JOIN Sales.Orders AS b

ON a.OrderID = b.OrderID

JOIN Sales.Customers AS c

ON c.CustomerID = b.CustomerID

WHERE b.OrderDate = @OrderDate

GROUP BY c.CustomerID, b.OrderID, b.OrderDate

IF @Exist > 0

RAISERROR('Exist', 16, 1)

COMMIT TRAN

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

BEGIN

PRINT ERROR\_MESSAGE()

ROLLBACK TRAN

END

END CATCH

END

GO

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.

create table ods.StockItem (

StockItemID int,

StockItemName nvarchar(100),

SupplierID int,

ColorID int,

UnitPackageID int,

OuterPackageID int,

Brand nvarchar(50),

Size nvarchar(20),

LeadTimeDays int,

QuantityPerOuter int,

IsChillerStock bit,

Barcode nvarchar(50),

TaxRate decimal(18,3),

UnitPrice decimal(18,2),

RecommendedRetailPrice decimal(18,2),

TypicalWeightPerUnit decimal(18,3),

MarketingComments nvarchar(MAX),

InternalComments nvarchar(MAX),

CountryOfManufature nvarchar(20)

);

insert into ods.StockItem

select StockItemID, StockItemName, SupplierID,ColorID,

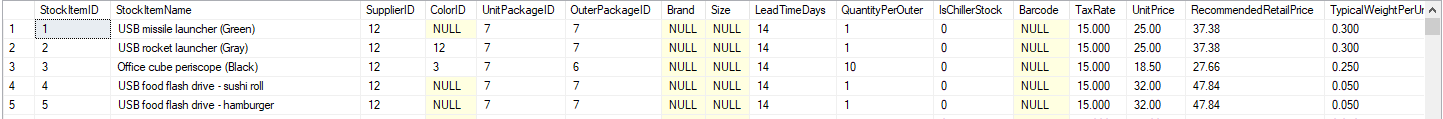
UnitPackageID,OuterPackageID,Brand,Size,LeadTimeDays,QuantityPerOuter,IsChillerStock,

Barcode,TaxRate,UnitPrice,RecommendedRetailPrice,TypicalWeightPerUnit,MarketingComments,InternalComments,

JSON\_value (wsi.CustomFields, '$.CountryOfManufacture')

from Warehouse.StockItems as wsi;

select \* from ods.StockItem



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 PROC sp21\_order @OrderDate Date

AS

BEGIN

DECLARE @Exist AS INT

SET @Exist = (SELECT Count(\*) from ods.Orders

WHERE OrderDate = @OrderDate) --check if @orderdate arleady existed in the target table

BEGIN TRY

BEGIN TRAN

INSERT INTO ods.Orders(CustomerID, OrderID, OrderDate, Total)

SELECT c.CustomerID, b.OrderID, b.OrderDate,

SUM((a.UnitPrice \* a.Quantity)) AS Total

FROM Sales.OrderLines AS a

JOIN Sales.Orders AS b

ON a.OrderID = b.OrderID

JOIN Sales.Customers AS c

ON c.CustomerID = b.CustomerID

WHERE b.OrderDate BETWEEN DATEADD(DAY, 1, @OrderDate) AND DATEADD(DAY, 7, @OrderDate)

GROUP BY c.CustomerID, b.OrderID, b.OrderDate

IF @Exist > 0

RAISERROR('Exist', 16, 1)

COMMIT TRAN

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

BEGIN

PRINT ERROR\_MESSAGE()

ROLLBACK TRAN

END

END CATCH

END

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, '$.PurchaseOrders')

WITH (

StockItemID int '$.StockItemID',

StockItemName nvarchar(100) '$.StockItemName',

SupplierID int '$.SupplierID',

UnitPackageID int '$.UnitPackageID',

OuterPackageID int '$.OuterPackageID',

Brand nvarchar(50) '$.Brand',

LeadTimeDays int '$.LeadTimeDays',

QuantityPerOuter int '$.QuantityPerOuter',

TaxRate decimal(18,3) '$.TaxRate',

UnitPrice decimal(18,3) '$.UnitPrice',

RecommendedRetailPrice decimal(18,2) '$.RecommendedRetailPrice',

TypicalWeightPerUnit decimal(18,3) '$.TypicalWeightPerUnit',

CustomFields nvarchar(max) '$' AS JSON ,

OrderDate date '$.OrderDate',

DeliveryMethodID int '$.DeliveryMethodID',

ExpectedDeliveryDate date '$.ExpectedDeliveryDate',

SupplierReference nvarchar(20) '$.SupplierReference'

);

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

CREATE VIEW stockitem\_qty\_bygroup25

AS

WITH SourceTable

AS

(SELECT SUM(a.Quantity) AS total, d.StockGroupName,

Year(b.OrderDate) AS yyear

FROM Sales.OrderLines as a

JOIN Sales.Orders AS b

ON b.OrderID = a.OrderID

JOIN Warehouse.StockItemStockGroups AS c

ON a.StockItemID = c.StockItemID

JOIN Warehouse.StockGroups AS d

ON c.StockGroupID = d.StockGroupID

GROUP BY d.StockGroupName, Year(b.OrderDate)

)

SELECT yyear, Clothing, [Computing Novelties], [Furry Footwear], Mugs,

[Novelty Items], [Packaging Materials], Toys, [T-Shirts], [USB Novelties]

FROM SourceTable

PIVOT (

MIN(total)

FOR StockGroupName IN (Clothing, [Computing Novelties], [Furry Footwear],

Mugs, [Novelty Items], [Packaging Materials], Toys, [T-Shirts],

[USB Novelties])) AS aa

GO

SELECT yyear AS [Year], Clothing, [Computing Novelties] , [Furry Footwear], [Mugs],

[Novelty Items],[Packaging Materials], [Toys], [T-Shirts], [USB Novelties]

FROM stockitem\_qty\_bygroup25

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.

CREATE VIEW stockitem\_qty\_bygroup26

AS

WITH SourceTable

AS

(SELECT SUM(a.Quantity) AS total, d.StockGroupName,

Year(b.OrderDate) AS yyear

FROM Sales.OrderLines as a

JOIN Sales.Orders AS b ON b.OrderID = a.OrderID

JOIN Warehouse.StockItemStockGroups AS c ON a.StockItemID = c.StockItemID

JOIN Warehouse.StockGroups AS d ON c.StockGroupID = d.StockGroupID

GROUP BY d.StockGroupName, Year(b.OrderDate)

)

SELECT yyear, Clothing, [Computing Novelties], [Furry Footwear], Mugs,

[Novelty Items], [Packaging Materials], Toys, [T-Shirts], [USB Novelties]

FROM SourceTable

PIVOT (

MIN(total)

FOR StockGroupName IN (Clothing, [Computing Novelties], [Furry Footwear],

Mugs, [Novelty Items], [Packaging Materials], Toys, [T-Shirts],

[USB Novelties])) AS aa

GO

SELECT

yyear AS [@Year], Clothing,[Computing Novelties] AS Computing\_Novelties,

[Furry Footwear]AS Furry\_Footwear, Mugs, [Novelty Items] AS Novelty\_Items,

[Packaging Materials] AS Packaging\_Materials, Toys, [T-Shirts] as t\_shirts,

[USB Novelties] AS USB\_Novelties

FROM stockitem\_qty\_bygroup26

FOR XML PATH('yyear')



28. Write a short essay talking about your understanding of transactions, locks and isolation levels.

A transaction is a logical unit of work that contains one or more SQL statements. It has four properties:

Atomicity: Each transaction is considered as one unit and either runs to completion or is not executed at all. There is no partially apply.

Consistency: Integrity constraints must be maintained so that the database is consistent before and after the transaction. It has to pass all the constraints to be valid.

Isolation: This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference.

Durability: This ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.

The ACID properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

Locks:

SQL Server locking is the essential part of the isolation requirement and it serves to lock the objects affected by a transaction. While objects are locked, SQL Server will prevent other transactions from making any change of data stored in objects affected by the imposed lock. Once the lock is released by committing the changes or by rolling back changes to initial state, other transactions will be allowed to make required data changes.

Exclusive lock: The exclusive lock will be imposed by the transaction when it wants to modify the page or row data. An exclusive lock can be imposed to a page or row only if there is no other shared or exclusive lock imposed already on the target.

Shared lock: This will reserve a row to be available only for reading, which means that any other transaction will be prevented to modify the locked record as long as the lock is active. However, a shared lock can be imposed by several transactions at the same time over the same row and in that way several transactions can share the ability for data reading since the reading process itself will not affect anyhow the actual row data.

Isolation level:

Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

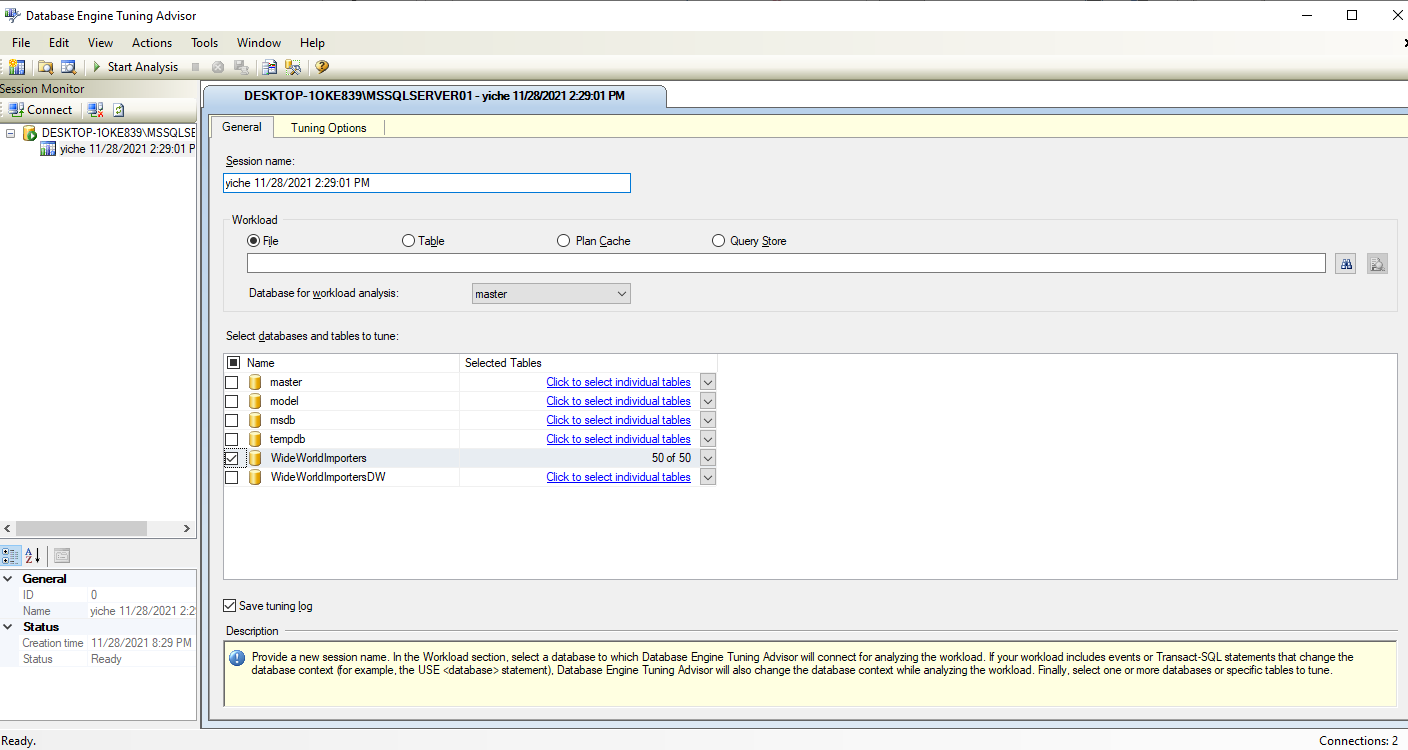
Read Committed guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

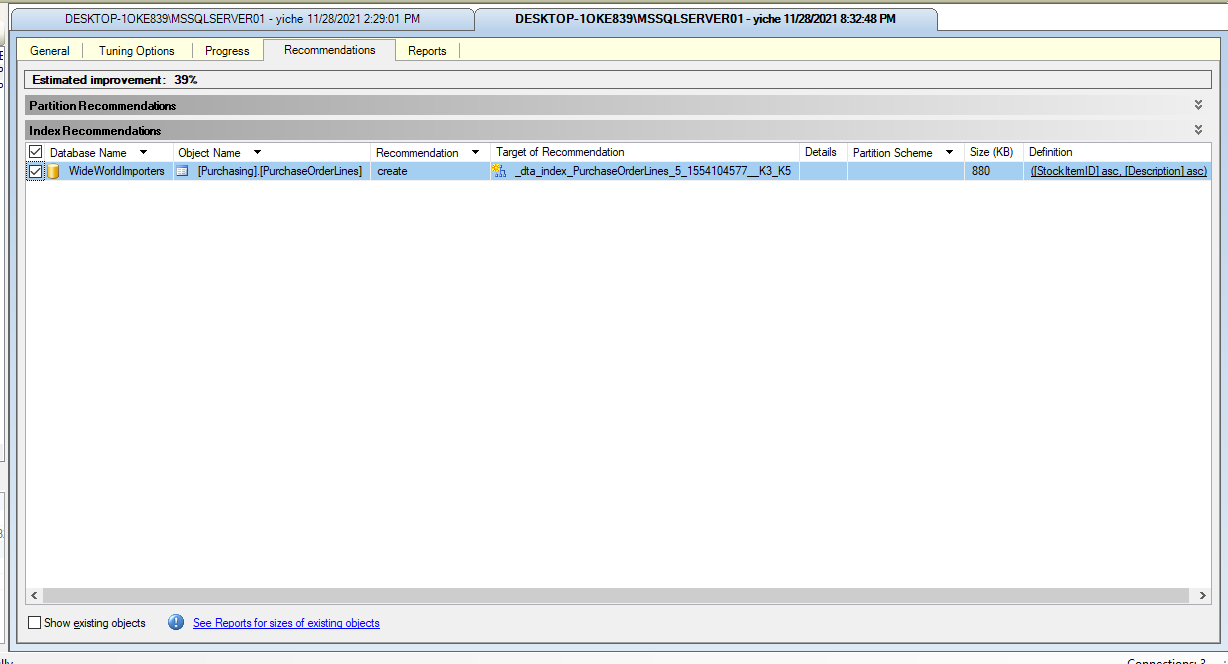
Repeatable Read is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

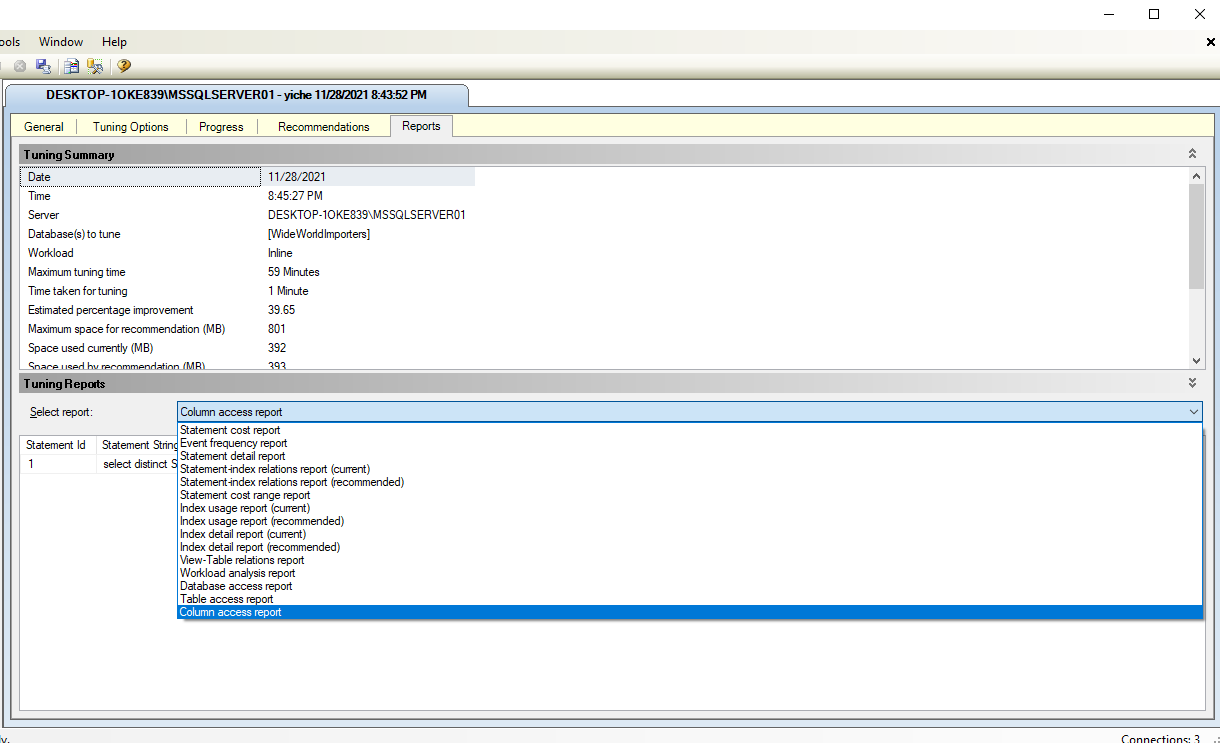
Serializable is the Highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

29. Write a short essay, plus screenshots talking about performance tuning in SQL Server. Must include Tuning Advisor, Extended Events, DMV, Logs and Execution Plan.

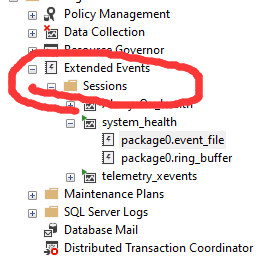
Tuning Advisor: tools – Database Engine Tuning Advisor

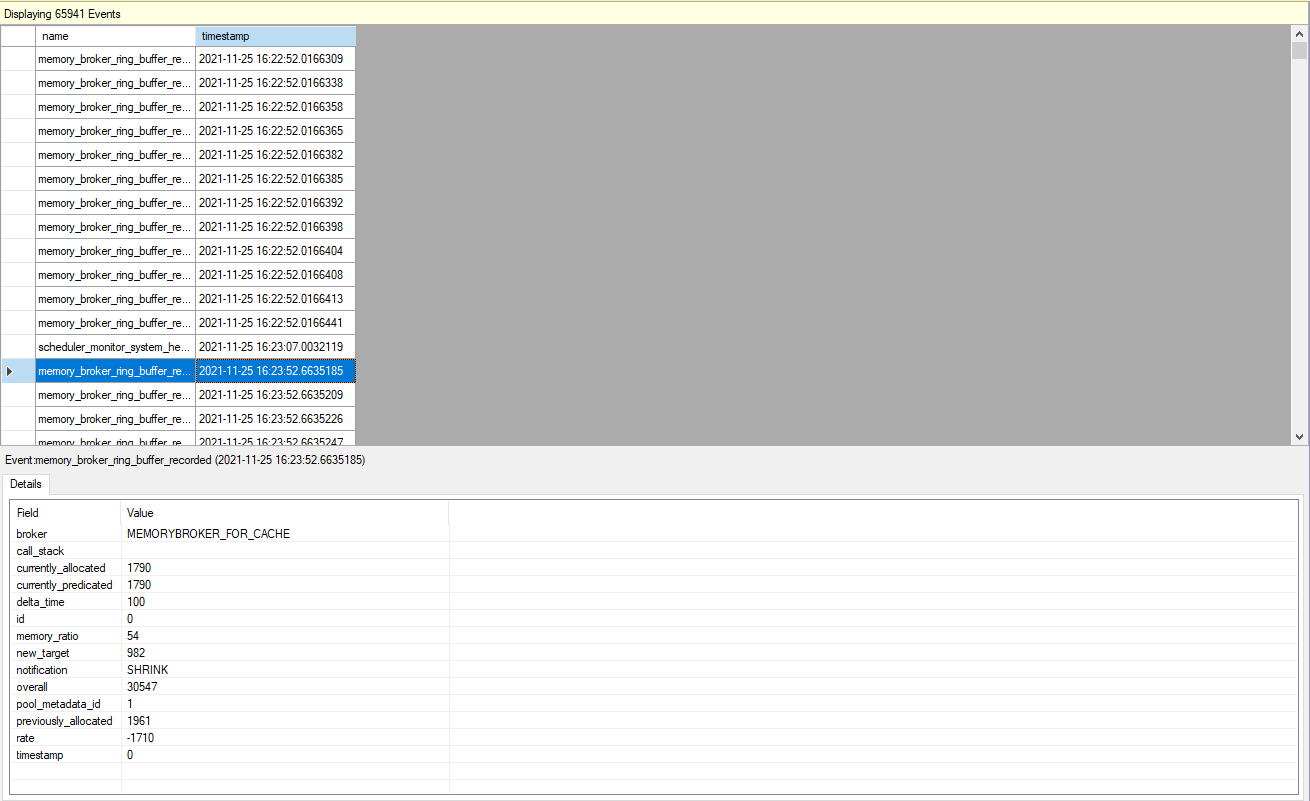






Extended Events



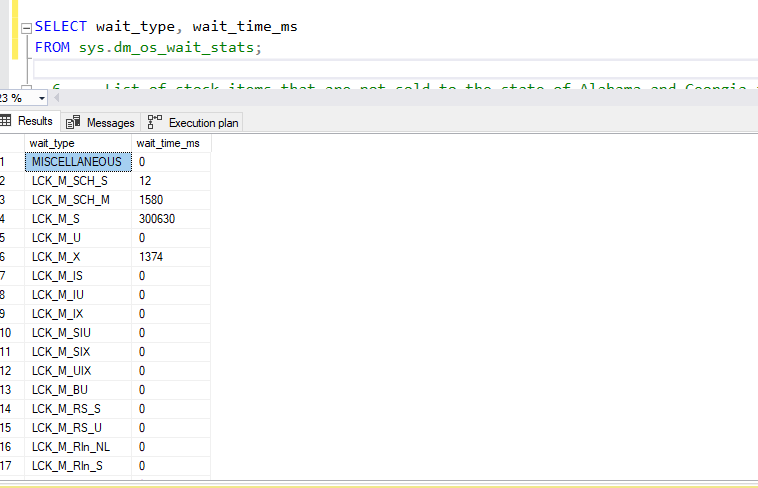


DMV: Dynamic management views and functions return server state information that can be used to monitor the health of a server instance, diagnose problems, and tune performance.

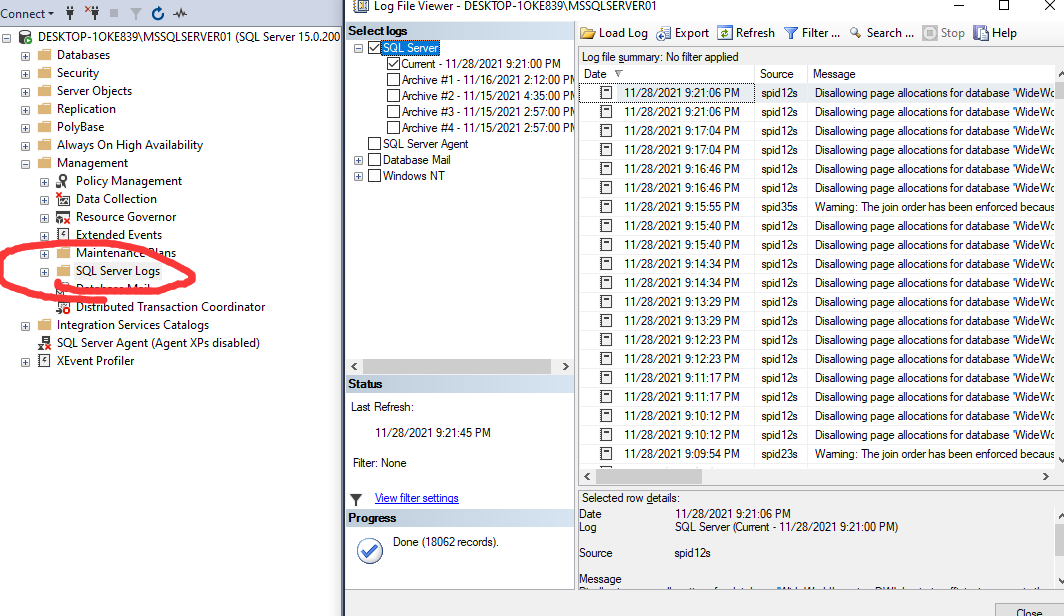
There are two types of dynamic management views and functions:

Server-scoped dynamic management views and functions. These require VIEW SERVER STATE permission on the server.

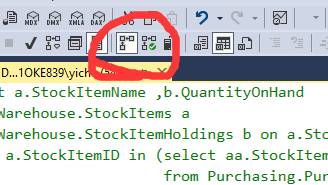
Database-scoped dynamic management views and functions. These require VIEW DATABASE STATE permission on the database.

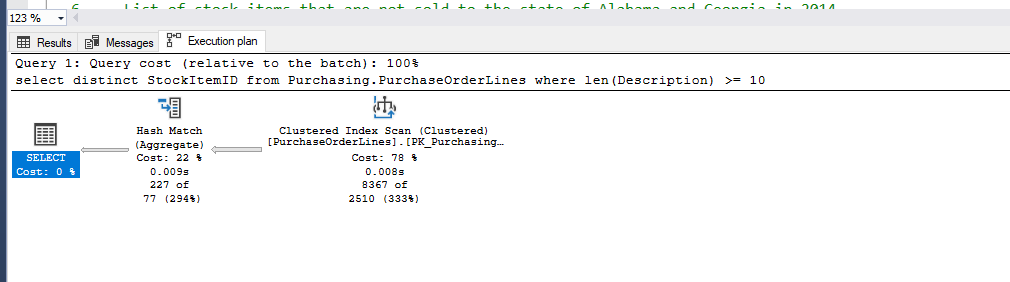


Log:



Execution Plan:





31. Database Design: OLTP db design request for EMS business: when people call 911 for medical emergency, 911 will dispatch UNITs to the given address. A UNIT means a crew on an apparatus (Fire Engine, Ambulance, Medic Ambulance, Helicopter, EMS supervisor). A crew member would have a medical level (EMR, EMT, A-EMT, Medic). All the treatments provided on scene are free. If the patient needs to be transported, that’s where the bill comes in. 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%). Bill should be sent to the patient insurance company first. If there is a deductible, we send the unpaid bill to the patient only. Don’t forget about patient information, medical nature and bill paying status.

