# SQL Assignments

SQL related assignments will be on the Wide World Importers Database unless otherwise mentioned.

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 p.FullName, p.FaxNumber as person_fax, p.PhoneNumber as
person_phone,c.CustomerName as company, c.PhoneNumber as company_phone, c.FaxNumber as
company_fax from
Application.People p join Sales.Customers c on
c.WebsiteURL LIKE '%'+ RIGHT(p.EmailAddress, CHARINDEX('@', REVERSE(p.EmailAddress))-
1) +'%'

UNION

select p.FullName, p.FaxNumber as person_fax, p.PhoneNumber as
person_phone,c.SupplierName as company, c.PhoneNumber as company_phone, c.FaxNumber as
company_fax from
Application.People p join Purchasing.Suppliers c on
c.WebsiteURL LIKE '%'+ RIGHT(p.EmailAddress, CHARINDEX('@', REVERSE(p.EmailAddress))-
1) +'%';
```

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

```
select c.CustomerID, c.CustomerName from
Sales.Customers c join Application.People p on
c.PrimaryContactPersonID = p.PersonID and p.PhoneNumber = c.PhoneNumber
;
```

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

```
select CustomerID from Sales.CustomerTransactions
where TransactionDate < Convert(datetime, '2016/01/01' )
group by CustomerID

EXCEPT

select CustomerID from Sales.CustomerTransactions
where TransactionDate >= Convert(datetime, '2016/01/01' )
group by CustomerID
;
```

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

```
select l.StockItemID, sum(l.ReceivedOuters) as quanitty from
Purchasing.PurchaseOrders o join Purchasing.PurchaseOrderLines l on
o.PurchaseOrderID = l.PurchaseOrderID and YEAR(o.OrderDate) = 2013
group by l.StockItemID
;
```

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

```
select StockItemID from Purchasing.PurchaseOrderLines
where LEN(Description) >= 10
;
```

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

```
with cities as (
    select c.CityID from
    Application.Cities c join Application.StateProvinces s on
    c.StateProvinceID = s.StateProvinceID and (s.StateProvinceName = 'Alabama' or
s.StateProvinceName = 'Georgia')
),
exclude as (
     select distinct xs.CustomerID from
     Warehouse.StockItemTransactions xs join Sales.Customers s on
     xs.CustomerID = s.CustomerID and YEAR(xs.TransactionOccurredWhen) = 2014)
,
orders as (
     select xs.StockItemID, xs.CustomerID from
     Warehouse.StockItemTransactions xs join Sales.Customers s on
     xs.CustomerID = s.CustomerID and xs.CustomerID not in (select * from exclude)
     group by xs.StockItemID, xs.CustomerID
)

select c1.StockItemID from
orders c1 join Sales.Customers c2 on
c1.CustomerID = c2.CustomerID and c2.DeliveryCityID not in (select * from cities)
;
```

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

```
with cities as (
    select c.CityID as city, s.StateProvinceID as state from
    Application.Cities c join Application.StateProvinces s on
    c.StateProvinceID = s.StateProvinceID
    group by c.CityID, s.StateProvinceID
)

select c2.state as StateID, avg(DATEDIFF(HOUR, t.TransactionDate,
i.ConfirmedDeliveryTime)) as avg_delivery_hours from
Sales.CustomerTransactions t join Sales.Invoices i
    on t.InvoiceID = i.InvoiceID
    join Sales.Customers c
            on i.CustomerID = c.CustomerID
            join cities c2
                    on c2.city = c.DeliveryCityID
group by c2.state
;
```

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

```
with cities as (
    select c.CityID as city, s.StateProvinceID as state from
    Application.Cities c join Application.StateProvinces s on
    c.StateProvinceID = s.StateProvinceID
    group by c.CityID, s.StateProvinceID
)

select c2.state as StateID, MONTH(t.TransactionDate) as month, avg(DATEDIFF(HOUR,
t.TransactionDate, i.ConfirmedDeliveryTime)) as avg_delivery_hours from
Sales.CustomerTransactions t join Sales.Invoices i
    on t.InvoiceID = i.InvoiceID
    join Sales.Customers c
            on i.CustomerID = c.CustomerID
            join cities c2
                    on c2.city = c.DeliveryCityID
group by c2.state, MONTH(t.TransactionDate)
order by c2.state, MONTH(t.TransactionDate)
;
```

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

```
with buy as (
    select l.StockItemID, sum(l.ReceivedOuters) as quantity from
    Purchasing.PurchaseOrders o join Purchasing.PurchaseOrderLines l on
    o.PurchaseOrderID = l.PurchaseOrderID and YEAR(o.OrderDate) = 2015
    group by l.StockItemID
),
sell as (
    select StockItemID, abs(total) as quantity from (
            select StockItemID, sum(Quantity) as total from
Warehouse.StockItemTransactions
            group by StockItemID) as t
    where total < 0
)
select b.StockItemID from
buy b left join sell s on b.StockItemID = s.StockItemID and b.quantity > s.quantity
;
```

10. 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 mugs as (
    select StockItemID as id from  Warehouse.StockItems
    where StockItemName like '%mug%'
),
exclude as (
select CustomerID from (
    select t.CustomerID, sum(i.Quantity) as total from
    Sales.CustomerTransactions t join Sales.InvoiceLines i on
    t.InvoiceID = i.InvoiceID and i.StockItemID in (select id from mugs) and
```

```
YEAR(t.TransactionDate) = 2016
    group by t.CustomerID) as cc
where cc.total > 10
)

select c.CustomerID, c.CustomerName, c.PhoneNumber, p.FullName as contact_name from
Sales.Customers c join Application.People p on
c.PrimaryContactPersonID = p.PersonID and c.CustomerID not in (select * from exclude)
group by c.CustomerID, c.CustomerName, c.PhoneNumber, p.FullName
;
```

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

```
select * from Application.Cities
where ValidFrom > Convert(datetime, '2015/01/01')
;
```

12. 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.

```
with cities as (
    select c.CityID as city, s.StateProvinceID as state, co.CountryID as country from
    Application.Cities c join Application.StateProvinces s on c.StateProvinceID =
s.StateProvinceID
            join Application.Countries co on co.CountryID = s.StateProvinceID
    group by c.CityID, s.StateProvinceID, co.CountryID
),
res as (
select
    w.StockItemName, (c.DeliveryAddressLine1 + ',' + c.DeliveryAddressLine2) as
delivery_address,
    ci.state, ci.city, ci.country, c.CustomerName, p.FullName as contact_person,
    c.PhoneNumber, l.Quantity
from
Sales.Orders o join Sales.OrderLines l on
o.OrderID = l.OrderID and o.OrderDate = Convert(datetime, '2014/07/01')
    join Warehouse.StockItems w on l.StockItemID = w.StockItemID
            join Sales.Customers c on o.CustomerID = c.CustomerID
                    join cities ci on ci.city = c.DeliveryCityID
                            join Application.People p on c.PrimaryContactPersonID =
p.PersonID
)

select * from res
;
```

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

```
with buy as (
    select s.StockGroupID, sum(p.OrderedOuters) as buy_quantity
    from Purchasing.PurchaseOrderLines p join Warehouse.StockItemStockGroups s on
p.StockItemID = s.StockItemID
```

```
    group by s.StockGroupID
),
sell as (
    select s.StockGroupID, sum(o.Quantity) as sell_quantity
    from Sales.OrderLines o join Warehouse.StockItemStockGroups s on o.StockItemID =
s.StockItemID
    group by s.StockGroupID
),
total as (
select b.StockGroupID, b.buy_quantity, s.sell_quantity from
buy b full outer join sell s on b.StockGroupID = s.StockGroupID
)
select s.StockGroupID, t.buy_quantity, t.sell_quantity, (t.buy_quantity -
t.sell_quantity) as remains from
Warehouse.StockItemStockGroups s left join total t on s.StockGroupID = t.StockGroupID
;
```

14. 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 delivery as (
    select l.StockItemID, c.DeliveryCityID, count(*) as delivery_count
    from Sales.Orders o join Sales.OrderLines l on o.OrderID = l.OrderID
            join Sales.Customers c on o.CustomerID = c.CustomerID
    where YEAR(o.OrderDate) = 2016
    group by l.StockItemID, c.DeliveryCityID
),
q as (
select StockItemID, DeliveryCityID,
    DENSE_RANK() over(partition by DeliveryCityID order by delivery_count desc) as
place
from delivery
),
city as(
    select DeliveryCityID, StockItemID
    from q where place = 1
)

select c.DeliveryCityID, ISNULL(s.StockItemName, 'No Sale') as StockItem
from city c join Warehouse.StockItems s on c.StockItemID = s.StockItemID
;
```

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

```
select * from
Sales.Invoices
where JSON_VALUE(ReturnedDeliveryData, '$.Events[1].Event') = 'DeliveryAttempt'
;
```

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

```
select StockItemID
from Warehouse.StockItems
where JSON_VALUE(CustomFields, '$.CountryOfManufacture') = 'China'
;
```

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

```
select sum(l.Quantity) as quantity, JSON_VALUE(s.CustomFields,
'$.CountryOfManufacture') as country
from sales.Orders o join Sales.OrderLines l on o.OrderID = l.OrderID
    join Warehouse.StockItems s on l.StockItemID = s.StockItemID
where YEAR(o.OrderDate) = 2015
group by JSON_VALUE(s.CustomFields, '$.CountryOfManufacture')
;
```

18. 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 Sales.P18 as
with total_data as (
    select YEAR(o.OrderDate) as [Year], sg.StockGroupName, sum(ol.Quantity) as quantity
    from Sales.Orders o join Sales.OrderLines ol on o.OrderID = ol.OrderID
            join Warehouse.StockItems s on ol.StockItemID = s.StockItemID
            join Warehouse.StockItemStockGroups g on g.StockItemID = s.StockItemID
            join Warehouse.StockGroups sg on g.StockGroupID = sg.StockGroupID
    where YEAR(o.OrderDate) BETWEEN 2013 AND 2017
    Group by YEAR(o.OrderDate), sg.StockGroupName
    )

select StockGroupName, [2013], [2014], [2015], [2016], [2017]
from total_data
pivot(
    sum(quantity) FOR
    Year IN ([2013], [2014], [2015], [2016], [2017])
) as pvt
;
```

19. 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 Sales.P19 as
with total_data as (
    select YEAR(o.OrderDate) as [Year], sg.StockGroupName, sum(ol.Quantity) as quantity
    from Sales.Orders o join Sales.OrderLines ol on o.OrderID = ol.OrderID
            join Warehouse.StockItems s on ol.StockItemID = s.StockItemID
            join Warehouse.StockItemStockGroups g on g.StockItemID = s.StockItemID
            join Warehouse.StockGroups sg on g.StockGroupID = sg.StockGroupID
    where YEAR(o.OrderDate) BETWEEN 2013 AND 2017
    Group by YEAR(o.OrderDate), sg.StockGroupName
    ),
groups as (
    select distinct StockGroupName from Warehouse.StockGroups
)

select [Year], [Novelty Items], [Clothing], [Mugs], [T-Shirts], [Airline Novelties],
[Computing Novelties],
           [USB Novelties], [Furry Footwear], [Toys], [Packaging Materials]
from total_data
pivot(
    sum(quantity)
```

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

20. 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 Sales.P20 (@id int)
returns TABLE as
return (
    select OrderID, Quantity * UnitPrice as Total
    from Sales.OrderLines where OrderID = @id
);

select * from
Sales.Invoices CROSS APPLY Sales.P20(OrderID);
```

21. 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 SCHEMA ods

GO

CREATE TABLE ods.Orders(
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    OrderTotal DECIMAL(18, 2),
    CustomerID INT
)
GO

CREATE PROCEDURE ods.DayTotal @date DATE AS

IF EXISTS (select 1 from ods.Orders where OrderDate = @date)
    BEGIN
            RAISERROR('Date Exists ', 16, 1)
    END
ELSE
    BEGIN
            BEGIN TRANSACTION
                    insert into ods.Orders
                    select o.OrderID, o.OrderDate, foo.Total, o.CustomerID
                    from Sales.Orders o cross apply Sales.P20(OrderID) foo
                    where o.OrderDate = @date
```

```
            COMMIT
    END
GO
EXEC ods.DayTotal '2015-01-01'
EXEC ods.DayTotal '2015-01-01'
EXEC ods.DayTotal '2015-01-02'
EXEC ods.DayTotal '2015-01-03'
EXEC ods.DayTotal '2017-01-01'
```

22. 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.StockItems(
    StockItemID INT PRIMARY KEY,
    StockItemName NVARCHAR(100) NOT NULL,
    SupplierID INT NOT NULL,
    ColorID INT NULL,
    UnitPackageID INT NOT NULL,
    OuterPackageID INT NOT NULL,
    Brand NVARCHAR(50) NULL,
    Size NVARCHAR(20) NULL,
    LeadTimeDays INT NOT NULL,
    QuantityPerOuter INT NOT NULL,
    IsChillerStock BIT NOT NULL,
    Barcode NVARCHAR(50) NULL,
    TaxRate DECIMAL(18, 3) NOT NULL,
    UnitPrice DECIMAL(18, 2) NOT NULL,
    RecommendedRetailPrice DECIMAL(18, 2) NULL,
    TypicalWeightPerUnit DECIMAL(18, 3) NOT NULL,
    MarketingComments NVARCHAR(MAX) NULL,
    InternalComments NVARCHAR(MAX) NULL,
    CountryOfManufacture NVARCHAR(20) NULL,
    [Range] NVARCHAR(20) NULL,
    Shelflife NVARCHAR(20) NULL
);

 INSERT INTO ods.StockItems
 SELECT
    StockItemID, StockItemName, SupplierID, ColorID, UnitPackageID, OuterPackageID,
 Brand, Size,
    LeadTimeDays, QuantityPerOuter, IsChillerStock, Barcode, TaxRate, UnitPrice,
 RecommendedRetailPrice,
    TypicalWeightPerUnit, MarketingComments, InternalComments,
    JSON_VALUE(CustomFields, '$.CountryOfManufacture'), JSON_VALUE(CustomFields,
 '$.Range'), JSON_VALUE(CustomFields, '$.ShelfLife')
 from Warehouse.StockItems
;
```

23. 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.

```
DROP PROCEDURE ods.DayTotal;

GO

CREATE PROCEDURE ods.DayTotal @date DATE AS

BEGIN TRANSACTION
    delete from ods.Orders where OrderDate < @date
COMMIT

BEGIN TRANSACTION
    MERGE ods.Orders o
    USING (
            select o.OrderID, o.OrderDate, foo.Total, o.CustomerID
            from Sales.Orders o CROSS APPLY Sales.P20(OrderID) foo
            where DATEDIFF(day, OrderDate, @date) BETWEEN 0 AND 7
            ) new_order
    ON o.OrderID = new_order.OrderID
    WHEN NOT MATCHED THEN INSERT VALUES (
    new_order.OrderID, new_order.OrderDate, new_order.Total, new_order.CustomerID);
COMMIT
```

24. 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.

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

```
select Year as Year,
    [Novelty Items] as 'StockGroup.Novelty Items',
    [Clothing] as 'StockGroup.Clothing',
    [Mugs] as 'StockGroup.Mugs',
    [T-Shirts] as 'StockGroup.T-Shirts',
    [Airline Novelties] as 'StockGroup.Airline Novelties',
    [Computing Novelties] as 'StockGroup.Computing Novelties',
    [USB Novelties] as 'StockGroup.USB Novelties',
    [Furry Footwear] as 'StockGroup.Furry Footwear',
    [Toys] as 'StockGroup.Toys',
    [Packaging Materials] as 'StockGroup.Packaging Materials'
from Sales.P19
FOR JSON PATH
```

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

```
select Year as '@Year',
    [Novelty Items] as NoveltyItems,
    [Clothing] as Clothing,
    [Mugs] as Mugs,
    [T-Shirts] as TShirts,
    [Airline Novelties] as AirlineNovelties,
    [Computing Novelties] as ComputingNovelties,
    [USB Novelties] as USBNovelties,
    [Furry Footwear] as FurryFootwear,
    [Toys] as Toys,
    [Packaging Materials] as PackagingMaterials
from Sales.P19
FOR XML PATH('StockItems')
```

27. 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.
28. Write a short essay talking about your understanding of transactions, locks and isolation levels.

Transaction is the basic client-side operation that can either be saved or undone. Locks are mechanisms that prevent conflicting writes or wrong reads to the same objects. And isolation levels describe kinds of possible concurrency issues of the DB, which means different isolation levels present different usage of locks and different performance.

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.

To perform performance tuning, one must first investigate current issues or bugs. In this case of monitoring, DMV offers administrative views that help us analyze table attributes and possibly historical records of queries so we can get some general ideas about what went wrong. And Extended Events offers a large set of events that we can monitor, where ideally, we would setup our monitoring events, execute the program, and go to extended events to see detailed logs of each event involved. Execution Plan is also useful in terms of monitoring the cost and the entire flow of our program, helping us grasping a general idea of what step to optimize. Then, we can use Tuning Advisor to configure our program by setting various strategies such as indexing, physical structures, partitions, thus finding out the missing optimizations. Last but not least, we can trace system logs or adding debugging messages using DBCC to see if there are significant errors and performance weakness.

Assignments 30 - 32 are group assignments.

30. Write a short essay 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.

    To migrate the user logon data, we need to first concat strings to form a selection of attributes as in Application.People: 1. Assign new and unique personID 2. Concat name, email address, password 3. Set most new columns to 0 (IsPermittedToLogon, IsExternalLogonProvider, IsSystemUser, IsSalesperson) and others to 1.

```
DECLARE @id INT;

select @id = MAX(personID) from Application.People

INSERT INTO Application.People
SELECT
p.BusinessEntityID + @id as PersonID, p.FirstName + ' ' + p.LastName as FullName,
p.FirstName as PreferredName, 0 as IsPermittedToLogon, e.EmailAddress as LogonName,
0 as IsExternalLogonProvider, pass.PasswordHash as HashedPassword, 0 as IsSystemUser,
phone.PhoneNumber as PhoneNumber,e.EmailAddress AS EmailAddress, 1 as LastEditedBy,

CASE WHEN emp.BusinessEntityID IS NOT NULL THEN 1 ELSE 0 END AS IsEmployee,
CASE WHEN s.BusinessEntityID IS NOT NULL THEN 1 ELSE 0 END AS IsSalesperson

FROM AdventureWorks.Person.Person p left join AdventureWorks.Person.EmailAddress e on
p.BusinessEntityID = e.BusinessEntityID
   left join AdventureWorks.Person.Password pass on p.BusinessEntityID =
pass.BusinessEntityID
   left join AdventureWorks.HumanResources.Employee emp on p.BusinessEntityID =
emp.BusinessEntityID
   left join AdventureWorks.Sales.SalesPerson s on p.BusinessEntityID =
s.BusinessEntityID
   left join AdventureWorks.Person.PersonPhone phone on p.BusinessEntityID =
phone.BusinessEntityID
;
```

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.

Schema 911

```
Unit {
        Id: primary key,
        UnitType: VARCHAR,
        Charge: int
}
CrewMember {
        Id: primary key,
        Medical level: TreatmentID, FK
}
Treatment {
        Id: primary key,
        Type: VARCHAR,
        Charge: int
}
Patient {
        Id: primary key,
        Insurance company: InsuranceID, FK,
        Address: VARCHAR,
        Contact: VARCHAR
}
Insurance {
        Id: primary key,
        Address: VARCHAR,
        Contact: VARCHAR
}
TransportMethod {
        Id: primary key,
        TransportType: VARCHAR,
        Charge: int
}
BillReceiver {
        Id: primary key,
        Patient: FK,
```

Insurance: FK
}
Bill {
Id: primary key,
UnitID: FK,
Transported: bool,
TransportedMiles: int,
TransportMethod: FK,
Patient: FK,
Deductible: bool,
BillReceiver: FK,
PayingStatus : VARCHAR
}

32. Remember the discussion about those two databases from the class, also remember, those data models are not perfect. You can always add new columns (but not alter or drop columns) to any tables. Suggesting adding Ingested DateTime and Surrogate Key columns. 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:
    a. Luckly, we only start with 1 fact: Purchase. Other facts can be ignored for now.
    b. Add a new dimension: Country of Manufacture. It should be given on top of Stock Items.
    c. Write script(s) and stored procedure(s) for the entire ETL from WWI db to DW.