Loading data from csv file into a dataframe using Spark

```
df = spark.read.option("header","true").format("csv").load("Files/MtoMActual.csv")

df2 =
spark.read.csv("abfss://FabricWorkspace@onelake.dfs.fabric.microsoft.com/DemoLakehouse.Lakehouse/Files/MtoMActual.csv")

df2 =
spark.read.load("abfss://FabricWorkspace@onelake.dfs.fabric.microsoft.com/DemoLakehouse.Lakehouse/Files/MtoMActual.csv",
format='csv', header=True)
```

Loading data from csv file into a dataframe using Pandas

```
import pandas as pd
# Load data into pandas DataFrame from "/lakehouse/default/" + "Files/MtoMActual.csv"
df = pd.read_csv("/lakehouse/default/Files/MtoMActual.csv")
display(df)
```

Saving data from a dataframe to a csv file or table

```
df.write.mode("overwrite").format("csv").save("Files/MtoMActual2.csv")

df.write.mode("overwrite").format("delta").saveAsTable("MtoMActual")
```

Loading data from a table

```
df = spark.read.table("Mtomactual")
display(df)

df2 = spark.read.format("delta").load("Tables/mtomactual")
```

```
display(df2)
```

```
%%sql
SELECT *
FROM mtomactual
```

Other ways to display data

```
df.collect()
df.schema
df.summary()
df.show()
```

Reducing the number of columns shown

```
dfreduced = df.select("Country", "Actual") # This is not case sensitive.
display(dfreduced)

dfreduced = df.select(df.Country, df.Actual) # This IS case sensitive.
display(dfreduced)

dfreduced = df.select(df.Country, df.Actual.alias("ActualSales"))
display(dfreduced)
```

```
%%sql
SELECT Country, Actual
FROM mtomactual; -- Semicolon needed if you have two statements in the one cell.

SELECT Country, Actual AS `Actual Sales` -- Need to use backticks if you are including a space. The “AS” is optional.
FROM mtomactual;
```

Filter data with a simple “where”

```
df = spark.read.table("Mtomactual")
dfreduced = df.select(df.Country, df.Actual.alias("ActualSales"))
display(dfreduced.where("ActualSales > 10000"))           # Use >, <, >=, <=, = or ==, != or <>
display(dfreduced.filter(dfreduced.Country == "England")) # Use >, <, >=, <=,      ==, !=.      Cannot use = or <>
display(dfreduced.limit(3))
display(dfreduced.tail(2))
```

```
%%sql
SELECT Country, Actual AS ActualSales
FROM mtomactual
WHERE Actual > 10000
LIMIT (2) -- cannot use TOP(2)
```

Adding additional columns

```
df = spark.read.table("Mtomactual")
display(df.withColumn("ActualDoubled", df.Actual * 2)\
          .withColumn("SecondLetter", df.Country.substr(2,1))
          )
```

```
from pyspark.sql.functions import *
df2 = df.withColumn("ColDate", add_months(lit("2028-01-01"), df.Actual/1000))
display(df2)
df2.write.mode("overwrite").format("delta").saveAsTable("MtoMActualWithDates")

df = spark.read.table("mtomactualwithdates")
display(df)
```

```
%%sql
SELECT *, Actual * 2 AS ActualDoubled, SUBSTR(Country, 2, 1) AS SecondLetter
```

```
FROM mtomactual;

SELECT *
FROM mtomactualwithdates
```

Advanced Filtering

```
df = spark.read.table("mtomactualwithdates")
display(df.where(df.Actual.between(6000, 11000)))
display(df.where(df.Country.contains("n")))
display(df.where(df.Country.like("%n_")))
display(df.where("Country = 'England' OR Country = 'France'"))
display(df.where( (df.Country == 'England') |  (df.Country == 'France')) )    # | = OR,    & = AND,    ~ = NOT
display(df.where(df.Country.isin("England", "France")))
```

```
%%sql
SELECT *
FROM mtomactualwithdates
WHERE Actual BETWEEN 6000 AND 11000
WHERE Country LIKE '%n_' -- %=0, 1, or more characters,   _ = 1 character.
WHERE Country = "England" OR Country = "France"
WHERE Country IN ("England", "France")
```

Convert data types

```
df = spark.read.table("mtomactual")
df = df.select(df.Country, df.Location, df.Actual.cast("int"))
display(df)
display(df.describe(["Country", "Actual"]).show())
```

```
%%sql
SELECT Country, Location, CAST(Actual as int)
```

```
FROM mtomactual
```

Importing data using a different data structure

```
from pyspark.sql.types import *
schemaTarget = StructType([StructField('Country', StringType(), True),
                           StructField('Location', StringType(), True),
                           StructField('Actual', IntegerType(), True)])
df = spark.read.option("header","true").format("csv").schema(schemaTarget).load("Files/MtoMActual.csv")
display(df)
df.schema
df.write.format("delta").saveAsTable("mtomactualstruct")

df = spark.read.table("mtomactualstruct")
display(df)
df.schema
```

Formatting dates as strings

```
from pyspark.sql.functions import *
df = spark.read.table("mtomactualwithdates")
df = df.select(df.Country, df.Location, df.Actual.cast("int"), \
               concat(lit("The date is: "),date_format(df.ColDate, "EEEE d MMMM yyyy")).alias("FormattedDate")
               )
display(df)
```

```
%%sql
SELECT Country, Location, CAST(Actual as int), concat("The date is: ",date_format(ColDate, "EEEE d MMMM yyyy")) AS
FormattedDate
FROM mtomactualwithdates
```

Grouping and Re-filtering data

```
df = spark.read.table("mtomactualstruct")
display(df.groupBy("Country").sum("Actual")\
          .withColumnRenamed("sum(Actual)","ActualTotal")\
          .where("ActualTotal>10000")
)
```

```
%%sql
SELECT Country, SUM(Actual) AS ActualTotal
FROM mtomactualstruct
GROUP BY Country
HAVING SUM(Actual) > 10000 -- ActualTotal
```

Sorting the results

```
df = spark.read.table("mtomactualstruct")
# display(df.orderBy("Location"))
# display(df.orderBy(df.Location))
# display(df.orderBy(asc(df.Location)))
# display(df.orderBy(desc(df.Country), df.Location))
# display(df.sort(desc("Country"), "Location"))
# display(df.sort("Country", ascending=False))
display(df.sort(df.Country.desc(), df.Location.asc()))
```

```
%%sql
SELECT *
FROM mtomactualstruct
--ORDER BY Location ASC
ORDER BY Country DESC, Location
```

Using all 6 SQL clauses

```
df = spark.read.table("mtomactualstruct")
display(df.select("Country", "Actual")\
          .where("Actual>4000")\
          .groupBy("Country").sum("Actual")\
          .withColumnRenamed("sum(Actual)","ActualTotal")\
          .where("ActualTotal>10000")
          .orderBy(desc("ActualTotal"),"Country")
          )
```

```
%%sql
SELECT Country, SUM(Actual) AS ActualTotal
FROM mtomactualstruct
WHERE Actual > 4000
GROUP BY Country
HAVING SUM(Actual) > 10000
ORDER BY ActualTotal DESC, Country
```

31a. Merging data

```
df = spark.read.table("mtomactual")
dfdates = spark.read.table("mtomactualwithdates")
dfadditional = spark.read.table("mtomactualadditional")
display(df)
display(dfadditional)
display(dfdates)

dfunion = df.union(dfadditional)
display(dfunion)
dfunion.write.mode("overwrite").format("delta").saveAsTable("MtoMActualCombined")
```

```
%%sql
SELECT *
FROM mtomactual
UNION -- OR UNION ALL
SELECT *
FROM mtomactualadditional
```

```
dfunion = df.unionByName(dfdates, allowMissingColumns=True)
display(dfunion)
```

```
%%sql
SELECT Country, Location, Actual, NULL as ColDate
FROM mtomactual
UNION ALL
SELECT Country, Location, Actual, ColDate
FROM mtomactualwithdates
```

32a. Identifying and resolving duplicate data

```
df = spark.read.table("mtomactualcombined")
display(df.distinct())
```

```
%%sql
SELECT DISTINCT Country, Location, Actual
FROM mtomactualcombined
```

```
display(df.groupBy("Country","Location","Actual").count().where("count>1"))
```

```
%%sql
SELECT Country, Location, Actual
FROM mtomactualcombined
GROUP BY Country, Location, Actual
```

```
HAVING COUNT(*)>1
```

```
display(df.dropDuplicates(["Country"]))
```

31b. Joining data

```
dfactual = spark.read.table("mtomactual")
dftarget = spark.read.format("csv").option("header","true").load("Files/MtoMTarget.csv")
dftarget.write.mode("overwrite").format("delta").saveAsTable("MtoMTarget")
display(dfactual)
display(dftarget)

dfactual = dfactual.select(dfactual.Country, dfactual.Actual.cast("int")) \
                   .groupBy("Country").sum("Actual").withColumnRenamed("sum(Actual)","ActualTotal")
display(dfactual)
dftarget = dftarget.select(dftarget.Country, dftarget.Target.cast("int")) \
                   .groupBy("Country").sum("Target").withColumnRenamed("sum(Target)","TargetTotal")
display(dftarget)

dfactual.write.mode("overwrite").format("delta").saveAsTable("MtoMActualSum")
dftarget.write.mode("overwrite").format("delta").saveAsTable("MtoMTargetSum")

dfjoin = dfactual.join(dftarget, dfactual.Country == dftarget.Country)
display(dfjoin)

display(dfactual.join(dftarget, "Country"))
```

```
%%sql
SELECT MtoMActualSum.Country, ActualTotal,
       MtoMTargetSum.Country, TargetTotal
FROM MtoMActualSum
```

```
FULL JOIN MtoMTargetSum
ON MtoMActualSum.Country = MtoMTargetSum.Country
```

```
dfactual = dfactual.withColumnRenamed("Country", "ActualCountry")
dftarget = dftarget.withColumnRenamed("Country", "TargetCountry")
dfjoin = dfactual.join(dftarget, dfactual.ActualCountry == dftarget.TargetCountry, "full")
display(dfjoin)
dfjoin.write.mode("overwrite").format("delta").saveAsTable("MtoMJoin")
```

```
%%sql
SELECT MtoMActualSum.Country AS ActualCountry, ActualTotal,
       MtoMTargetSum.Country AS TargetCountry, TargetTotal
FROM MtoMActualSum
FULL JOIN MtoMTargetSum
ON MtoMActualSum.Country = MtoMTargetSum.Country
```

32b. Resolving missing data or null values

```
display(dfjoin.where(dfjoin.ActualTotal.isNull()))
```

```
SELECT MtoMActualSum.Country AS ActualCountry, ActualTotal,
       MtoMTargetSum.Country AS TargetCountry, TargetTotal
FROM MtoMActualSum
FULL JOIN MtoMTargetSum
ON MtoMActualSum.Country = MtoMTargetSum.Country
WHERE ActualTotal IS NULL
```

```
dfjoin = dfjoin.fillna({"ActualCountry":"(No Actual)"})
dfjoin = dfjoin.fillna({"TargetCountry":"(No Target)"})
dfjoin = dfjoin.na.fillna(0)
```

```
SELECT IFNULL(MtoMActualSum.Country, "(No Actual)") AS ActualCountry, COALESCE(ActualTotal, 0) AS ActualTotal,
       IFNULL(MtoMTargetSum.Country, "(No Target)") AS TargetCountry, COALESCE(TargetTotal, 0) AS TargetTotal
```

```
FROM MtoMActualSum
FULL JOIN MtoMTargetSum
ON MtoMActualSum.Country = MtoMTargetSum.Country
```

Practice Activity – Spark

```
dfactual = spark.read.table("mtomactualsum")
dftarget = spark.read.table("mtomtargetsum")
display(dfactual)
display(dftarget)

dfbridge = dfactual.select("Country")
dfbridge = dfbridge.union(dftarget.select("Country")).distinct()
display(dfbridge)

display(dfbridge.join(dfactual, "Country", "left").join(dftarget, "Country", "left"))
```

31. Process data by using Spark structured streaming

Step 1 – retrieve the schema for the Lakehouse table

```
df = spark.read.format("csv").option("header","true").load("Files/Shopping.csv")
df.schema
```

Step 2 – load the stream with the specified schema

```
from pyspark.sql.types import StructType, StructField, StringType

schema = StructType([
    StructField("Date", StringType(), True),
    StructField("Subtype", StringType(), True),
    StructField("ShopaseMethod", StringType(), True),
    StructField("Out", StringType(), True)
])

dfs = spark.readStream.option("header", "true").schema(schema).format("csv").load("Files/Shop*.csv")
```

Step 3 – Write the deduplicated stream to a Delta Lake table

```
dfs = spark.readStream.option("header", "true").schema(schema).format("csv").load("Files/Shop*.csv")
deltatablepath = "Tables/Shop_table"
query = dfs.writeStream.format("delta").outputMode("append") \
        .option("checkpointLocation", deltatablepath + '/_checkpoint') \
        .option("path", deltatablepath).start()
```

Step 4 – Wait for the streaming query to finish

```
query.awaitTermination()
```

Final version for the Spark Job Definition

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType
spark = SparkSession.builder.getOrCreate()

schema = StructType([StructField('Date', StringType(), True),
StructField('Subtype', StringType(), True),
StructField('PurchaseMethod', StringType(), True),
StructField('Out', StringType(), True)])

dfs = spark.readStream.option("header", "true").schema(schema).format("csv").load("Files/Shop*.csv")
deltatablepath = "Tables/Shop_table"
query = dfs.writeStream.format("delta").outputMode("append") \
        .option("checkpointLocation", deltatablepath + '/_checkpoint') \
        .option("path", deltatablepath).start()
query.awaitTermination()
```

18. Slowly Changing Dimensions (SCD)

Query 1 – Creating the tables

```
DROP TABLE IF EXISTS FactImport

CREATE TABLE FactImport(
ID INT,
OrderDate DATE,
ProductID INT,
Cost DECIMAL(7,2))

DROP TABLE IF EXISTS DimensionImport

CREATE TABLE DimensionImport(
ProductID INT,
Name VARCHAR(20),
UpdateDate DATE)

DROP TABLE IF EXISTS FactOverall

CREATE TABLE FactOverall(
ID INT,
OrderDate DATE,
ProductID INT,
Cost DECIMAL(7,2))

DROP TABLE IF EXISTS DimensionOverall

CREATE TABLE DimensionOverall(
ProductID INT,
Name VARCHAR(20),
UpdateDate DATE
, StartDate DATE
, EndDate DATE
, IsCurrent CHAR(1)
)
```

Query 2 – Adding data and running the stored procedure

```
-- Empty Import tables
DELETE FROM FactImport
DELETE FROM DimensionImport

-- Import new data
INSERT INTO FactImport(ID, OrderDate, ProductID, Cost) VALUES
(1, '2024-01-02', 1, 34),
(2, '2024-01-03', 2, 48),
(3, '2024-02-02', 1, 60),
(4, '2024-02-03', 2, 23),
(5, '2024-03-02', 1, 76),
(6, '2024-03-03', 2, 12),
(7, '2024-04-02', 1, 95),
(8, '2024-04-03', 2, 34)

INSERT INTO DimensionImport(ProductID, Name, UpdateDate) VALUES
(1, 'Product 1', '2023-12-31'),
(2, 'Product 2', '2023-12-31'),
(1, 'Product 1', '2024-01-31'),
(2, 'Product 2', '2024-01-31'),
(1, 'Product 1a', '2024-02-29'),
(2, 'Product 2', '2024-02-29'),
(1, 'Product 1a', '2024-03-31'),
(2, 'Product 2', '2024-03-31'),
(1, 'Product 1b', '2024-04-30'),
(2, 'Product 2', '2024-04-30')

DELETE FROM FactImport      WHERE MONTH(OrderDate)  <> 12 -- Change to 1, 2, 3, and 4
DELETE FROM DimensionImport WHERE MONTH(UpdateDate) <> 12 -- Change to 1, 2, 3, and 4

EXECUTE SCD0 -- Change to SCD1 and SCD2
```

Query 3 – Slowly Changing Dimensions Type 0

```
DROP PROCEDURE IF EXISTS SCD0
GO

CREATE PROC SCD0 AS
BEGIN
  INSERT INTO DimensionOverall(ProductID, Name, UpdateDate)
  SELECT ProductID, Name, UpdateDate
  FROM DimensionImport
  WHERE ProductID NOT IN (SELECT ProductID FROM DimensionOverall)

  INSERT INTO FactOverall(ID, OrderDate, ProductID, Cost)
  SELECT ID, OrderDate, ProductID, Cost
  FROM FactImport
  WHERE ID NOT IN (SELECT ID FROM FactOverall)

  SELECT * FROM FactOverall ORDER BY ID
  SELECT ProductID, Name FROM DimensionOverall ORDER BY UpdateDate, ProductID
END
```

Query 4 – Slowly Changing Dimensions Type 1

```
DROP PROCEDURE IF EXISTS SCD1
GO

CREATE PROC SCD1 AS
BEGIN
  UPDATE DimensionOverall
  SET DimensionOverall.Name = DimensionImport.Name, DimensionOverall.UpdateDate = DimensionImport.UpdateDate
  FROM DimensionImport
  LEFT JOIN DimensionOverall
  ON DimensionImport.ProductID = DimensionOverall.ProductID
  WHERE DimensionImport.ProductID IN (SELECT ProductID FROM DimensionOverall)
    AND (DimensionImport.Name <> DimensionOverall.Name)

  INSERT INTO DimensionOverall(ProductID, Name, UpdateDate)
  SELECT ProductID, Name, UpdateDate
  FROM DimensionImport
  WHERE ProductID NOT IN (SELECT ProductID FROM DimensionOverall)

  INSERT INTO FactOverall(ID, OrderDate, ProductID, Cost)
  SELECT ID, OrderDate, ProductID, Cost
  FROM FactImport
  WHERE ID NOT IN (SELECT ID FROM FactOverall);

  SELECT * FROM FactOverall ORDER BY ID
  SELECT ProductID, Name, UpdateDate FROM DimensionOverall ORDER BY UpdateDate, ProductID
END
```

Query 5 – Slowly Changing Dimensions Type 2

```
DROP PROCEDURE IF EXISTS SCD2
```

```
GO

CREATE PROC SCD2 AS
BEGIN
    UPDATE DimensionOverall
    SET EndDate = (Select Max(UpdateDate) from DimensionOverall), IsCurrent = 'X'
    FROM DimensionImport
    LEFT JOIN DimensionOverall
    ON DimensionImport.ProductID = DimensionOverall.ProductID
    WHERE DimensionImport.ProductID IN (SELECT ProductID FROM DimensionOverall)
     AND (DimensionImport.Name <> DimensionOverall.Name)
     AND EndDate IS NULL;

    INSERT INTO DimensionOverall(ProductID, Name, UpdateDate, StartDate, IsCurrent)
    SELECT DISTINCT ProductID, Name, UpdateDate, dateadd(day,1,(Select Max(UpdateDate) from DimensionOverall WHERE
DimensionOverall.ProductID = DimensionImport.ProductID)), 'Y'
    FROM DimensionImport
    WHERE ProductID IN (SELECT ProductID FROM DimensionOverall WHERE DimensionImport.ProductID =
DimensionOverall.ProductID AND DimensionImport.Name <> DimensionOverall.Name AND DimensionOverall.IsCurrent IN ('Y',
'X'));

    UPDATE DimensionOverall
    SET IsCurrent = 'N'
    FROM DimensionImport
    WHERE IsCurrent = 'X'

    UPDATE DimensionOverall
    SET UpdateDate = DimensionImport.UpdateDate
    FROM DimensionImport
    LEFT JOIN DimensionOverall
    ON DimensionImport.ProductID = DimensionOverall.ProductID
    WHERE DimensionImport.ProductID IN (SELECT ProductID FROM DimensionOverall)
    AND IsCurrent = 'Y';

    INSERT INTO DimensionOverall(ProductID, Name, UpdateDate, StartDate, IsCurrent)
    SELECT ProductID, Name, UpdateDate, '2010-01-01', 'Y'
    FROM DimensionImport
    WHERE ProductID NOT IN (SELECT ProductID FROM DimensionOverall);

    INSERT INTO FactOverall(ID, OrderDate, ProductID, Cost)
    SELECT ID, OrderDate, ProductID, Cost
    FROM FactImport
    WHERE ID NOT IN (SELECT ID FROM FactOverall);

    SELECT ID, OrderDate, FactOverall.ProductID, Name as ProductName, Cost
    FROM FactOverall
    LEFT JOIN DimensionOverall
    ON FactOverall.ProductID = DimensionOverall.ProductID
    AND OrderDate >= StartDate AND OrderDate <= ISNULL(EndDate, '2099-01-01')
    ORDER BY ID;

    SELECT ProductID, Name, UpdateDate, StartDate, EndDate, IsCurrent FROM DimensionOverall ORDER BY UpdateDate,
ProductID;
END
```

Creating tables in a Data Warehouse

```
DROP TABLE IF EXISTS tblTarget

CREATE TABLE tblTarget
(
Country VARCHAR(20),
Type VARCHAR(20),
Target INT
)

DROP TABLE IF EXISTS tblActual

CREATE TABLE tblActual
(
Country VARCHAR(20),
Location VARCHAR(20),
Actual INT
)
```

Inserting data into tables and transforming data in a Data Warehouse

```
DROP TABLE IF EXISTS tblTarget

CREATE TABLE tblTarget
(
Country VARCHAR(20),
Type VARCHAR(20),
Target INT
)

DROP TABLE IF EXISTS tblActual

CREATE TABLE tblActual
(
Country VARCHAR(20),
Location VARCHAR(20),
Actual INT
)

INSERT INTO tblActual (Country, Location, Actual) VALUES
('England', 'London', 5000),
('England', 'Birmingham', 7000),
('England', 'Manchester', 11000),
('France', 'Paris', 4000),
('Italy', 'Milan', 3000),
('Italy', 'Rome', 13000);

INSERT INTO tblTarget (Country, Type, Target) VALUES
('England', 'In Store', 10000),
('England', 'Internet/Post', 5000),
('France', 'In Store', 7500),
('France', 'Internet/Post', 3000),
('Germany', 'In Store', 8000),
('Germany', 'Internet/Post', 4000);

SELECT Country, SUM(Actual) AS TotalActual
INTO tblActualSum
FROM tblActual
GROUP BY Country

SELECT Country, SUM(Target) as TotalTarget
INTO tblTargetSum
FROM tblTarget
GROUP BY Country

SELECT * FROM [DemoWarehouse].[dbo].[tblActualSum]
SELECT * FROM [DemoWarehouse].[dbo].[tblTargetSum]
```

Implement file partitioning

```
df = spark.read.table("AddressData")
display(df)

df = spark.read.parquet("Tables/AddressData")
display(df)

df = spark.read.parquet("Tables/AddressData/CountryRegion=Canada")
display(df)

df = spark.read.option("recursiveFileLookup","true").parquet("Tables/AddressData/CountryRegion=Canada/*/*.parquet")
display(df)

df = spark.read.parquet("Tables/AddressData/CountryRegion=Canada/StateProvince=Alberta")
display(df)
```

```
df = spark.read.table("DimProduct")
df.write.mode("overwrite").format("delta").partitionBy("Color").saveAsTable("DimProductPartition")
df.write.mode("overwrite").format("delta").partitionBy("Color").save("Tables/DimProductPartition")
```

Creating views, stored procedures and functions

```
CREATE VIEW dbo.view_AddressData2 AS
SELECT *
FROM [DemoLakehouse].[dbo].[AddressData]

SELECT *
FROM view_AddressData

CREATE PROC dbo.proc_AddressData @Country varchar(20) AS
BEGIN
```

```
SELECT *
FROM view_AddressData

SELECT *
FROM view_AddressData2
WHERE CountryRegion = @Country
END

EXEC dbo.proc_AddressData "Canada"

CREATE FUNCTION dbo.func_AddressData (@Country AS varchar(20))
RETURNS TABLE
AS
RETURN
SELECT *
FROM AddressData
WHERE CountryRegion = @Country

SELECT AddressID, City
FROM func_AddressData('Canada')
```

Improvement performance in notebooks

```
spark.conf.get('spark.sql.parquet.vorder.enabled')
spark.conf.set('spark.sql.parquet.vorder.enabled', 'true')
```

```
%%sql
SET spark.sql.parquet.vorder.enabled=TRUE
```

```
spark.conf.set('spark.microsoft.delta.optimizeWrite.enabled', 'true')
```

Data Loading bottlenecks in SQL queries

```sql
SELECT *
FROM [DemoLakehouse].[dbo].[FactInternetSales]
```

Performance improvements in SQL queries

```sql
SELECT mtomactualsum.Country AS ActualCountry, ActualTotal,
       mtomtargetsum.Country AS TargetCountry, TargetTotal
FROM mtomactualsum
FULL JOIN mtomtargetsum
ON mtomactualsum.Country = mtomtargetsum.Country

SELECT Country, Location, Actual
FROM mtomactual
WHERE Country = 'England'

SELECT Country, SUM(Actual) AS TotalActual
FROM mtomactualstruct
GROUP BY Country

SELECT Country, Location, Actual
FROM mtomactual
ORDER BY Country, Location

SELECT *
FROM [DemoLakehouse].[dbo].[FactInternetSales]
WHERE Year(OrderDate) = 2007

SELECT *
FROM [DemoLakehouse].[dbo].[FactInternetSales]
```

```
WHERE SUBSTRING(SalesOrderNumber, 1, 3) = 'SO5'
```

16. Implement orchestration patterns with notebooks and pipelines, including parameters and dynamic expressions

```
ParameterCountry = "England"
```

```
df = spark.read.table("Mtomactual")
df = df.where(df.Country == ParameterCountry)
display(df)
df.write.mode("overwrite").format("delta").saveAsTable("MtoMActualExtract")

df2 = spark.read.table("MtomactualExtract")
display(df2)
```

```
# Add Base parameters: Parameter Name = ParameterCountry; Type = String; Value = France or Italy
```

17. Incremental data loads – Data Warehouse to Lakehouse

Step 1 Query 1 – For Data Warehouse – Creating the tables

```
DROP TABLE IF EXISTS tblWaterMark;
DROP TABLE IF EXISTS FactImport;
DROP PROCEDURE IF EXISTS updateWatermark;

CREATE TABLE FactImport(
ID INT,
OrderDate DATETIME2(6), -- This has been changed to allow times
ProductID INT,
Cost DECIMAL(7,2))
```

```
CREATE TABLE tblWaterMark
(Watermark DATETIME2(6))

INSERT INTO tblWaterMark
VALUES ('2000-01-01')

GO

CREATE PROCEDURE updateWatermark @WaterMark DATETIME2(6)
AS
UPDATE tblWaterMark
SET Watermark = @WaterMark

SELECT * FROM tblWaterMark
EXEC updateWatermark '2000-01-02T00:00:00' -- To test
```

Step 1 Query 2 – For Data Warehouse – Adding data and running the stored procedure

```
-- Empty Import tables
DELETE FROM FactImport

-- Import new data
INSERT INTO FactImport(ID, OrderDate, ProductID, Cost) VALUES
(1, '2024-01-02', 1, 34),
(2, '2024-01-03', 2, 48),
(3, '2024-02-02', 1, 60),
(4, '2024-02-03', 2, 23),
(5, '2024-03-02', 1, 76),
(6, '2024-03-03', 2, 12),
(7, '2024-04-02', 1, 95),
(8, '2024-04-03', 2, 34)
```

```
DELETE FROM FactImport      WHERE MONTH(OrderDate)  IN (2, 3, 4) -- Expand by removing , 1, 2, 3, and 4
SELECT * FROM FactImport
```

Step 3 – For Pipeline – Lookup Activity

```
SELECT MAX(OrderDate) AS LatestWatermark
FROM FactImport
```

Step 4 – For Pipeline – Copy data Activity

```
SELECT * FROM FactImport WHERE OrderDate > '@{activity('LookuptblWatermark').output.firstRow.Watermark}'
                          AND OrderDate <= '@{activity('MaximumWatermark').output.firstRow.LatestWatermark}'
```

Step 5 – For Pipeline – Stored procedure Activity

```
@{activity('MaximumWatermark').output.firstRow.LatestWatermark}
```

Step 6 – For Notebook – Query

```
df = spark.read.csv("Files/Incremental/*.txt", header=True)
display(df)
```

17. Using a dataflow to implement incremental data loads

```
DROP TABLE IF EXISTS [SalesLT].[NewTable];
CREATE TABLE [SalesLT].[NewTable]
(ID INT null,
Name VARCHAR(50) NULL,
UpdateDate DATE NULL);

INSERT INTO [SalesLT].[NewTable]
VALUES (1, 'First row', GETDATE()-1),
       (2, 'Second row', GETDATE()-1);
```

```
SELECT * FROM [SalesLT].[NewTable];

INSERT INTO [SalesLT].[NewTable]
VALUES (3, 'Third row', GETDATE());

UPDATE [SalesLT].[NewTable]
SET Name = 'Updated row', UpdateDate = GETDATE()
FROM [SalesLT].[NewTable]
WHERE ID = 2;

SELECT * FROM [SalesLT].[NewTable];
```

Query converted from SQL

```
--
explain
SELECT TOP 10 State, COUNT(*) AS NumberOfRows
FROM Weather
GROUP BY State
ORDER BY NumberOfRows

Weather
| summarize NumberOfRows=toint(count()) by State
| project State, NumberOfRows
| sort by NumberOfRows desc nulls first
| take int(10)
```

Selecting data

```
Weather
```

```
| project State, EpisodeId, StartDate=StartTime, EndTime, EpisodeNarrative, EventNarrative
| extend Duration = EndTime-StartDate
| project-away EventNarrative
// | extend EpisodeNarrative = "hi"
//
| project-rename Narrative = EpisodeNarrative
| order by State asc nulls last
```

Limiting the number of rows (slide 159)

```
You can end a query with a semicolon.
You must end a query with a semicolon to separate multiple queries.
Weather | take 10
Weather | limit 10 // run it multiple times – the same response is given.
Weather | sample 10 // CAN retrieve different rows
```

```
Weather
| distinct State
| limit 10

Weather
| sample-distinct 10 of State
```

```
Weather
| distinct State
| top 10 by State // default by descending
```

```
Weather
| distinct State
| top 10 by State asc nulls last
```

Filtering data - Where

String literals

```
print 'hello';
print "hello";
print 'She said "Hello"'
print "She said 'Hello'"
```

```
print "A backslash is this symbol: \. It is used as a special character."
// does not work
print "A backslash is this symbol: \t. It is used as a special character." // tab
print "A backslash is this symbol: \n. It is used as a special character." // new line
print "A backslash is this symbol: \u0021. It is used as a special character." // unicode
```

```
print ```She said
A blackslash is \\
Hello.```               // a multi-line string literal
```

Comparing the entirety of strings

```
Weather
| project Narrative=EpisodeNarrative, EventType
| where EventType == "Heavy Rain"  // case sensitive

Weather
| project Narrative=EpisodeNarrative, EventType
| where EventType =~ "heavy rain" // case insensitive

Weather
| project Narrative=EpisodeNarrative, EventType
| where EventType != "Heavy Rain"  // case sensitive
```

```
Weather
| project Narrative=EpisodeNarrative, EventType
| where EventType !~ "heavy rain" // case insensitive
```

```
Weather
| project Narrative=EpisodeNarrative, EventType
| where not (EventType == "Heavy Rain")  // case sensitive
```

```
Weather
| project Narrative=EpisodeNarrative, EventType
| where EventType == "Heavy Rain" or EventType == "Blizzard"  // case sensitive

Weather
| project Narrative=EpisodeNarrative, EventType, State
| where State == "TEXAS" and (EventType == "Heavy Rain" or EventType == "Blizzard")
// case sensitive

Weather
| project Narrative=EpisodeNarrative, EventType
| where EventType in ("Heavy Rain", "Blizzard")  // case sensitive
```

```
Weather
| project Narrative=EpisodeNarrative, EventType
| where not(EventType =~ "heavy rain" or EventType =~ "blizzard")  // case insensitive

Weather
| project Narrative=EpisodeNarrative, EventType
| where EventType !in~ ("heavy rain", "blizzard")  // case insensitive
```

Comparing part of strings

```
Weather
| project EpisodeNarrative
| where EpisodeNarrative has "county"  // case insensitive

Weather
| project EpisodeNarrative
| where EpisodeNarrative contains "county"  // case insensitive

Weather
| project EpisodeNarrative
| where EpisodeNarrative has_cs "county"  // case sensitive
```

```
Weather
| project EpisodeNarrative
| where EpisodeNarrative has "cou"

Weather
| project EpisodeNarrative
| where EpisodeNarrative contains "cou"
```

```
Weather
| project EpisodeNarrative
| where EpisodeNarrative hasprefix "cou"
| where not (EpisodeNarrative hasprefix "count")
| where EpisodeNarrative !hasprefix "couple"
```

```
Weather
| project EpisodeNarrative
| where EpisodeNarrative startswith "showers"
```

```
Weather
| project EpisodeNarrative
| where EpisodeNarrative has "showers" or EpisodeNarrative has "county"

Weather
| project EpisodeNarrative
| where EpisodeNarrative has "showers" and EpisodeNarrative has "county"

Weather
| project EpisodeNarrative
| where EpisodeNarrative has_any ("showers", "county")

Weather
| project EpisodeNarrative
| where EpisodeNarrative has_all ("showers", "county")
```

Aggregating data

```
Weather
| summarize Calc = count() by State, EventType

Weather
| summarize Calc = countif(EventType == "Flood") by State, EventType

Weather
| summarize Calc = countif(EventType != "Flood") by State, EventType

Weather
| where State in~ ("Texas", "Kansas", "Alaska")
| summarize Calc = countif(EventType =~ "flood") by State, EventType
| where Calc >= 100
```

```
Weather
| summarize Calc = sum(InjuriesDirect) by State

Weather
| summarize Calc = sumif(InjuriesDirect, EventType == "Flood") by State

Weather
| summarize Calc = avg(InjuriesDirect) by State

Weather
| summarize Calc = max(InjuriesDirect) by State

Weather
| summarize Calc = min(InjuriesDirect) by State

Weather
| summarize Calc = maxif(InjuriesDirect, EventType == "Flood") by State

Weather
| summarize Calc = dcount(EventType) by State

Weather
| summarize Calc = count_distinct(EventType) by State
```

String functions

Empty strings (note: also can use isnotempty, isnull and isnotnull)

```
Weather
| project State, EpisodeNarrative, EventType, BeginLocation
| where isempty(BeginLocation) // or == ""
| limit 20
```

Combining strings together and trimming the result

```
Weather
| project State, EpisodeNarrative, EventType, BeginLocation
| extend Calc = strcat(State, ": ", EventType, ": ", EpisodeNarrative)
| limit 20

Weather
| project State, EpisodeNarrative, EventType, BeginLocation
| extend Calc = strcat_delim(": ", State, EventType, EpisodeNarrative)
| limit 20

Weather
| project State, EpisodeNarrative, EventType, BeginLocation, EventId
| extend EventType = toupper(EventType) // and tolower
| extend Calc = strcat_delim(": ", EventId, State, EventType, EpisodeNarrative)
| limit 20

Weather
| project State, BeginLocation, EndLocation
| extend Calc = trim(' ', strcat_delim(" ", BeginLocation, State, EndLocation))
| limit 20
```

Substring and strlen

```
Weather
| project EventType
| extend FindSpace = indexof(EventType, " ") // gives -1 if there is not a space
| limit 20
| extend FindSpace = indexof(strcat(EventType, " "), " ")
```

```
, BeforeSpace = substring(EventType, 0, 5) // zero-based
```

```
| extend FindSpace = indexof(strcat(EventType, " "), " ")
, BeforeSpace = substring(EventType, 0, FindSpace) // a problem, as FindSpace has not yet been defined.
```

```
| extend FindSpace = indexof(strcat(EventType, " "), " ")
| extend BeforeSpace = substring(EventType, 0, FindSpace)
| project FindSpace, BeforeSpace, State, EpisodeNarrative, EventType, BeginLocation
```

```
| extend AfterSpace = substring(EventType, FindSpace+1, 999)
| project FindSpace, BeforeSpace, AfterSpace, State, EpisodeNarrative, EventType, BeginLocation
```

```
| extend NumCharactersNeeded = strlen(EventType)-FindSpace-1
| extend AfterSpace = substring(EventType, FindSpace+1, NumCharactersNeeded)
```

replace_string

```
| extend EventType = replace_string(EventType, "heavy", "huge") // does not change “Heavy”.
| extend EventType = replace_string(EventType, "Heavy", "Huge")
```

Mathematical, rounding functions

Data types

```
Weather
| project EpisodeId, EventId
| extend Calc = EpisodeId + EventId
| limit 20

| extend Calc = EpisodeId - EventId
| extend Calc = EpisodeId - EventId*5
| extend Calc = EpisodeId - EventId/5 //returns a number/long
| extend Calc = EpisodeId - EventId/5.0 //returns a real – note the .x99999 numbers.

| extend Calc = EpisodeId - EventId / int(5)
| extend Calc = EpisodeId - EventId / long(5)
| extend Calc = EpisodeId - EventId / real(5)
| extend Calc = EpisodeId - EventId / decimal(5) // exact division.

| extend Calc = round(EpisodeId - EventId/ real(5), 0) // and ,1) for 1 decimal place
| extend Calc = ceiling(EpisodeId - EventId/ real(5), 0) rounds up
```

Other math functions

```
| extend Calc2 = abs(Calc)

| extend Calc3 = sign(Calc)

| extend Calc = EpisodeId % (EventId/10) // delete Calc2 and Calc3.

| extend Calc = pow(EpisodeId, 2)

| extend Calc = sqrt(EpisodeId)

| extend Calc = rand()
| extend Calc = rand(4)
// int     - signed 32-bit integer
// long    - signed 64-bit integer
// real    - signed 64-bit double-precision, floating-point number
// decimal - signed 128-bit decimal number.
```

datetime/timespan functions

The datetime and timespan data types

```
Weather
| distinct StartTime, EndTime
| extend Duration = EndTime - StartTime, RevisedEndTime = EndTime + 1d
// d h m s ms microsecond tick

print datetime(2030-02-03 01:23:45.6);
print todatetime("2030-02-03 01:23:45.6");
print make_datetime(2030, 2, 3, 1, 23, 45.6);

print timespan(1d);
print timespan(3);
print timespan(40 seconds);
print timespan(1.23:45:17.8);
print timespan(1.23:45:17.8) - 3h;

print timespan(1.23:45:17.8) * 3;

print timespan(1.23:45:17.8) / 3h;

print now() + 3h;
print now(3h)
print now(-3h)
print ago(3h)
```

Extracting parts of dates

```
Weather
| distinct StartTime
| where StartTime between (datetime(2007-03-26) .. datetime(2007-04-15))

| extend Calc = startofday(StartTime) // and endofday

Weather
| distinct StartDate = startofday(StartTime)
| extend Calc = startofweek(StartDate) // start of the week = Sunday.
| where StartDate between (datetime(2007-03-26) .. datetime(2007-04-15))
// also startofmonth, startofyear, endofweek, endofmonth, endofyear

| extend Calc = dayofweek(StartDate)
// and also hourofday, dayofmonth, dayofyear, weekofyear, monthofyear, getyear

| extend Calc = datetime_part("week_of_year", StartDate)
// start of the week = Sunday. First week includes the first Thursday.
// you can extract Year, Quarter, Month, Day, DayOfYear, Hour, Minute, Second, Millisecond, Microsecond and Nanosecond
```

Date manipulation

```
Weather
| distinct StartTime
| where StartTime between (datetime(2007-03-26) .. datetime(2007-04-15))

| extend Calc = StartTime+2d
, Calc2= datetime_add('day', 2, StartTime)
// The period can be Year, Quarter, Month, Week, Day, Hour, Minute, Second, Millisecond, Microsecond and Nanosecond

| extend Calc = EndTime-StartTime
, Calc2= datetime_diff('day', EndTime, StartTime)
// datetime_diff only uses the relevant period.
// 26th 00:00 to 26th 23:00 is 0 days
// 26th 14:00 to 27th 08:00 is 1 day
```

```
Weather
| distinct StartTime
| extend Calc = datetime_utc_to_local(StartTime, 'US/Eastern')
| where StartTime between (datetime(2007-03-26) .. datetime(2007-04-15))

| extend Calc = datetime_local_to_utc(StartTime, 'US/Eastern')

print timezones = datetime_list_timezones()
| mv-expand timezones // expands a list into records
```

Converting dates and timestamps

```
Weather
| distinct StartTime, EndTime
| extend Calc = format_datetime(StartTime, 'd MM yyyy')
,       Calc2 = format_timespan(EndTime - StartTime, 'ddd.hh:mm:ss')
```

Format specifier	Data Type	Description
`y or yy`	datetime	The year, from 0 (or 00) to 99.
`yyyy`	datetime	The year as a four-digit number.
`M and MM`	datetime	The month, from 1 (or 01) through 12.
`d and dd`	datetime	The day of the month, from 1 (or 01) through 31.
`d to dddddddd`	timespan	Number of days, with extra zeros if needed.
`h and hh`	datetime	The hour, using a 12-hour clock from 1 (or 01) to 12.
`H and HH`	Both	The hour, using a 24-hour clock from 0 (or 00) to 23.
`m and mm`	Both	The minute, from 0 (or 00) through 59.
`s or ss`	Both	The second, from 0 (or 00) through 59.

f to fffffff	Both	Fractions of a second in a date and time value.
F to FFFFFFF	Both	If non-zero, fractions of a second in a date and time value.
tt	datetime	AM / PM hours

```
// Delimiters: space / - : , . _ [ and ].
// Note: there is no dddd (for “Tuesday”) or mmmm (for “January”).
```

Unioning tables together

```
Weather
| distinct State, EventType
| union kind = outer withsource = TableSource
(Weather
| distinct State, EventNarrative)

union withsource = AdditionalColumn Wea*
```

Joining tables together

```
Weather
| join
(Region | extend State = toupper(State))
on State

Weather
| join kind = fullouter
(Region | extend State = toupper(State))
on ($left.State == $right.State)

Weather
```

```
| lookup kind = leftouter
(Region | extend State = toupper(State))
on ($left.State == $right.State)
```

Identify and resolve duplicate data, missing data, or null values

```
Weather
| join kind = fullouter
(Region | extend State = toupper(State))
on ($left.State == $right.State)
| project State, Region
| summarize Count = count() by State, Region
| where Count > 1

Weather
| join kind = fullouter
(Region | extend State = toupper(State))
on ($left.State == $right.State)
//| where State1 == "" or isnull(State1)
| where coalesce(State1, "") == ""

Weather
| join kind = rightanti
(Region | extend State = toupper(State))
on ($left.State == $right.State)
```

Conditional functions

Iif (or Iff – they work the same)

```
Weather
```

```
| summarize NumberOfEvents = count() by State
| extend TexasOrFlorida = iif(State == "TEXAS" or State == "FLORIDA", "Texas/Florida", "Other")

Weather
| summarize NumberOfEvents = count() by State
| extend TexasOrFlorida = iif(State in ("TEXAS", "FLORIDA"), "Texas/Florida", "Other")

Weather
| summarize NumberOfEvents = count() by State
| extend TexasOrFlorida = case(State == "TEXAS", "Texas/Florida", State == "FLORIDA", "Texas/Florida", "Other")

// the Else argument is required.
```

48. Optimize Spark performance

```
dfbridge.explain(True)
```

```
spark.sql("CACHE TABLE mtomactual")
```

```
spark.conf.get('spark.ms.autotune.enabled')
```

```
spark.conf.set('spark.ms.autotune.enabled', 'true')
```

```
spark.conf.get('spark.sql.shuffle.partitions')
```

```
spark.conf.get('spark.sql.autoBroadcastJoinThreshold')
```

```
spark.conf.get('spark.sql.files.maxPartitionBytes')
```

Implement dynamic data masking

```
DROP TABLE IF EXISTS DynamicDataMasking;
GO
CREATE TABLE DynamicDataMasking
(ID int                  MASKED WITH (FUNCTION = 'random(1,10)'),
EmailAddress varchar(50) MASKED WITH (FUNCTION = 'email()'),
FamilyName varchar(30)   MASKED WITH (FUNCTION = 'partial(1, "XXXXXX", 2)'),
Department varchar(30)   MASKED WITH (FUNCTION = 'default()'),
DateOfBirth date         MASKED WITH (FUNCTION = 'default()'))

INSERT INTO DynamicDataMasking (ID, EmailAddress, FamilyName, Department, DateOfBirth)
VALUES
(11, 'john.doe@company.com',              'Doe',     'IT',          '1985-02-15'),
(12, 'jane.smith@hrsolutions.org',        'Smith',   'HR',          '1990-06-10'),
(13, 'michael.brown@financeworld.net',    'Brown',   'Finance',     '1982-09-21'),
(14, 'lisa.white@marketinghub.co.uk',     'White',   'Marketing',   '1988-11-30'),
(15, 'david.johnson@salespro.io',         'Johnson', 'Sales',       '1992-01-25'),
(16, 'emily.taylor@engineeringplus.tech', 'Taylor',  'Engineering', '1995-05-14'),
(17, 'chris.lee@legalshield.us',          'Lee',     'Legal',       '1987-08-19'),
(18, 'sarah.wilson@itglobal.info',        'Wilson',  'IT',          '1989-12-07'),
(19, 'james.moore@hrcareers.biz',         'Moore',   'HR',          '1993-04-03'),
(20, 'anna.jackson@financemaster.de',     'Jackson', 'Finance',     '1986-07-09');

SELECT * FROM DynamicDataMasking

GRANT SELECT ON DynamicDataMasking TO [jane@Filecats.onmicrosoft.com]

ALTER TABLE DynamicDataMasking
ALTER COLUMN ID DROP MASKED;
ALTER TABLE DynamicDataMasking
ALTER COLUMN ID ADD MASKED WITH (FUNCTION = 'random(21, 30)')
```

Using an eventstream in a KQL Database

```
BikeTable
| summarize NoBikes = avg(left_No_Bikes) by left_BikepointID, left_Neighbourhood, right_Window_End_Time
| order by left_Neighbourhood asc, right_Window_End_Time asc
```

Revising KQL syntax

```
BikeTable
| distinct left_Neighbourhood, left_Street
| sort by left_Neighbourhood asc, left_Street
| limit 5

BikeTable
| project Neighbourhood = left_Neighbourhood, left_Street, left_BikepointIDNumber, left_No_Empty_Docks
| extend Location = strcat(Neighbourhood, " ", left_Street)
// | where Neighbourhood !~ 'strand'
| where left_BikepointIDNumber > 600 and left_Street contains 'a'

BikeTable
| summarize No_Empty_Docks = avg(left_No_Empty_Docks) by left_Neighbourhood
| sort by No_Empty_Docks

--
explain
SELECT AVG(left_No_Empty_Docks) AS No_Empty_Docks, left_Neighbourhood
FROM BikeTable
GROUP BY left_Neighbourhood
```

```
BikeTable
| summarize No_Empty_Docks=tolong(avg(left_No_Empty_Docks)) by left_Neighbourhood
| project No_Empty_Docks, left_Neighbourhood
```

Row-level security in a Data Warehouse

```
DROP TABLE IF EXISTS tblActual

CREATE TABLE tblActual
(
Country VARCHAR(20),
Location VARCHAR(20),
Actual INT,
UserName VARCHAR(20)
)

INSERT INTO tblActual (Country, Location, Actual, UserName) VALUES
('England', 'London', 5000, 'Susan'),
('England', 'Birmingham', 7000, 'Susan'),
('England', 'Manchester', 11000, 'Susan'),
('France', 'Paris', 4000, 'Jane'),
('Italy', 'Milan', 3000, 'Jane'),
('Italy', 'Rome', 13000, 'Thomas');

SELECT USER_NAME()

CREATE FUNCTION securityfunction(@UserName AS VARCHAR(50))
RETURNS TABLE
WITH SCHEMABINDING
AS
```

```
RETURN
SELECT 1 AS securityfunction
WHERE @UserName = LEFT(USER_NAME(), LEN(@Username)) OR LEFT(USER_NAME(), 4) = 'Jane'

CREATE SECURITY POLICY SecurityPolicy
ADD FILTER PREDICATE dbo.securityfunction(UserName)
ON dbo.tblActual
WITH (STATE = ON);

SELECT * FROM tblActual
```

Column-level security in a Data Warehouse

```
DROP TABLE IF EXISTS tblActual

CREATE TABLE tblActual
(
Country VARCHAR(20),
Location VARCHAR(20),
Actual INT,
UserName VARCHAR(20)
)

INSERT INTO tblActual (Country, Location, Actual, UserName) VALUES
('England', 'London', 5000, 'Susan'),
('England', 'Birmingham', 7000, 'Susan'),
('England', 'Manchester', 11000, 'Susan'),
('France', 'Paris', 4000, 'Jane'),
('Italy', 'Milan', 3000, 'Jane'),
('Italy', 'Rome', 13000, 'Thomas');
```

```
SELECT * FROM tblActual;

GRANT SELECT ON tblActual TO [jane@Filecats.onmicrosoft.com]
GRANT SELECT ON tblActual(Country, Location) TO [susan@Filecats.onmicrosoft.com]
```

Object-level security

```
GRANT SELECT ON tblActual TO [jane@Filecats.onmicrosoft.com]
GRANT SELECT ON tblActual(Country, Location) TO [susan@Filecats.onmicrosoft.com]
REVOKE SELECT ON tblActual TO [susan@Filecats.onmicrosoft.com]

DENY SELECT ON tblActual TO [Microsoft@Filecats.onmicrosoft.com]

SELECT * FROM sys.database_principals

GRANT SELECT ON tblActual TO [New Group 2]
```