Calculation items

```
DimDate = ADDCOLUMNS(CALENDARAUTO(),
"CalendarYear", Year([Date]),
"MonthNumber", MONTH([Date]),
"MonthName", FORMAT([Date], "MMMM"),
"Quarter", "Q" & QUARTER([Date]),
"DayOfWeekNumber", WEEKDAY([Date]),
"DayOfWeekName", FORMAT([Date], "dddd")
)


MTD = CALCULATE(SELECTEDMEASURE(), DATESMTD(DimDate[Date]))

QTD = CALCULATE(SELECTEDMEASURE(), DATESQTD(DimDate[Date]))

YTD = CALCULATE(SELECTEDMEASURE(), DATESYTD(DimDate[Date]))

PY = CALCULATE(SELECTEDMEASURE(), SAMEPERIODLASTYEAR(DimDate[Date]))

SalesAmountPYMTD = CALCULATE(SUM(FactInternetSales[SalesAmount]), SAMEPERIODLASTYEAR(DimDate[Date]),'Time
Intelligence'[Time Calculation] = "MTD")

PY MTD = CALCULATE(SELECTEDMEASURE(),SAMEPERIODLASTYEAR(DimDate[Date]),'Time Intelligence'[Time Calculation] = "MTD")

PY QTD = CALCULATE(SELECTEDMEASURE(),SAMEPERIODLASTYEAR(DimDate[Date]),'Time Intelligence'[Time Calculation] = "QTD")

PY YTD = CALCULATE(SELECTEDMEASURE(),SAMEPERIODLASTYEAR(DimDate[Date]),'Time Intelligence'[Time Calculation] = "YTD")

YOY = SELECTEDMEASURE() - CALCULATE(SELECTEDMEASURE(), 'Time Intelligence'[Time Calculation] = "PY")

YOY% = DIVIDE(CALCULATE(SELECTEDMEASURE(),'Time Intelligence'[Time Calculation]="YOY"),
CALCULATE(SELECTEDMEASURE(),'Time Intelligence'[Time Calculation]="PY"))
```

```
SalesAmountYOY% = CALCULATE([SumOfSalesAmount], 'Time Intelligence'[Time Calculation] = "YOY%")

Field chooser = {
    ("SalesAmount", NAMEOF('FactInternetSales'[SumOfSalesAmount]), 0),
    ("OrderQuantity", NAMEOF('FactInternetSales'[OrderQuantitySum]), 1),
    ("ExtendedAmount", NAMEOF('FactInternetSales'[ExtendedAmountSum]), 2)
}
```

Using DAX variables

```
SalesAmountCalculation =
VAR MonthToDate = CALCULATE(SUM(FactInternetSales[SalesAmount]),DATESMTD(FactInternetSales[DueDate]))
VAR CurrentSales = SUM(FactInternetSales[SalesAmount])
RETURN (MonthToDate - CurrentSales) / MonthToDate
```

ROWNUMBER

```
RowNumberColumn = ROWNUMBER(
    ALLSELECTED(DimProduct[EnglishProductSubcategoryName],DimProduct[EnglishProductCategoryName]),
    ORDERBY(DimProduct[EnglishProductSubcategoryName],ASC),
    PARTITIONBY(DimProduct[EnglishProductCategoryName]))

RowNumberColumn2 = ROWNUMBER(
    ALLSELECTED(DimProduct[EnglishProductSubcategoryName],DimProduct[EnglishProductCategoryName]),
    ORDERBY(DimProduct[EnglishProductCategoryName],ASC))
```

RANK

```
RankNumberColumnDense = RANK(DENSE,
    ALLSELECTED(DimProduct[EnglishProductSubcategoryName],DimProduct[EnglishProductCategoryName]),
    ORDERBY(DimProduct[EnglishProductCategoryName],ASC))
```

```
RankNumberColumnSkip = RANK(SKIP,
    ALLSELECTED(DimProduct[EnglishProductSubcategoryName],DimProduct[EnglishProductCategoryName]),
    ORDERBY(DimProduct[EnglishProductCategoryName],ASC))
```

INDEX

```
SummaryTable = SUMMARIZECOLUMNS(DimDate[CalendarYear], DimProduct[EnglishProductCategoryName], "SalesAmount",
SUM(FactInternetSales[SalesAmount]))

IndexCalculation = INDEX(2, ALL(SummaryTable[CalendarYear]), ORDERBY(SummaryTable[CalendarYear], DESC))
```

OFFSET

```
TheOtherYearSales = CALCULATE(SUM(SummaryTable[SalesAmount]), OFFSET(1, ,
ORDERBY(SummaryTable[EnglishProductCategoryName],ASC, SummaryTable[CalendarYear])))
```

WINDOW

```
RunningTotal = CALCULATE(
    sum(FactInternetSales[SalesAmount]),
    WINDOW(-1, REL, 0, REL, ORDERBY(DimDate[CalendarYear])))
```

Dynamic strings

```
if(MIN(MtoMTransactions[Currency])<>MAX(MtoMTransactions[Currency]),"Multiple",
if(MIN(MtoMTransactions[Currency])="USD","$#,##0",
if(MIN(MtoMTransactions[Currency])="EUR","€#,##0",
if(MIN(MtoMTransactions[Currency])="GBP","£#,##0",
MIN(MtoMTransactions[Currency]) & " #,##0"))))
```

Creating our first SQL query, and reducing the number of columns shown

```sql
SELECT *
FROM purchases

SELECT Date, Subtype, PurchaseMethod AS "Purchase Method"
FROM purchases
```

String functions

```sql
SELECT Subtype, PurchaseMethod, Subtype + ' ' + PurchaseMethod
FROM purchases

SELECT Subtype, PurchaseMethod, CONCAT(Subtype, ' ', PurchaseMethod, ' ', Subtype)
FROM purchases

SELECT Subtype, PurchaseMethod, CONCAT_WS(' ', Subtype, PurchaseMethod, Subtype)
FROM purchases

SELECT Subtype, LEFT(Subtype, 1), RIGHT(Subtype, 1), SUBSTRING(Subtype, 2, 3)
FROM purchases

SELECT Subtype, LEN(SUBSTRING(Subtype, 7, 1))
FROM purchases

SELECT Subtype, TRIM(SUBSTRING(Subtype, 6, 1) + Subtype + SUBSTRING(Subtype, 11, 1))
FROM purchases

SELECT Subtype, UPPER(Subtype)
FROM purchases
```

```
SELECT Subtype, REPLACE(Subtype, 'Misc', 'Miscellaneous')
FROM purchases
```

Mathematical functions

```
SELECT Out, Out+10, Out-10, Out*10, Out/10
FROM purchases

SELECT Out, SQUARE(Out), SQRT(Out), POWER(Out, 3)
FROM purchases

SELECT Out, FLOOR(Out), CEILING(Out), ROUND(Out, 0)
FROM purchases

SELECT Out, ROUND(Out-7.9, 2), SIGN(Out-7.9), ABS(Out-7.9)
FROM purchases
```

Date/Time functions

```
SELECT Date, DATEADD(DAY, 1, Date), DATEADD(MONTH, 1, Date), DATEADD(DAY, -1, Date)
FROM purchases

SELECT Date, DATETRUNC(MONTH, Date), EOMONTH(Date), DATEDIFF(DAY, DATETRUNC(MONTH, Date), EOMONTH(Date)) + 1
FROM purchases

SELECT Date, DAY(Date), MONTH(Date), YEAR(Date), DATENAME(WEEKDAY, Date), DATEPART(WEEKDAY, Date)
FROM purchases

SELECT CURRENT_TIMESTAMP, GETDATE(), SYSDATETIME()

SELECT Date, DATEDIFF(MONTH, CURRENT_TIMESTAMP, EOMONTH(Date))
```

```
FROM purchases

SELECT Date, DATEFROMPARTS(2026, 2, 3), DATEDIFF(MONTH, DATEFROMPARTS(2026, 2, 3), EOMONTH(Date))
FROM purchases

-- You can also use DATEFROMPARTS, DATETIME2FROMPARTS, DATETIMEFROMPARTS, SMALLDATETIMEFROMPARTS and TIMEFROMPARTS
```

Converting and formatting between date and string data types

```
SELECT Date, Subtype, Out
FROM purchases

SELECT 'Date is: ' + Date
FROM purchases

SELECT DAY(Date)
FROM purchases

SELECT Date, CAST(Date AS DATETIME2), CONVERT(DATETIME2, Date)
FROM purchases

SELECT TRY_CAST('31/02/2027' AS DATE)

SELECT PARSE('12/11/2027' AS DATE USING 'en-GB')

SELECT 'Date is: ' + FORMAT(CAST(Date AS DATE), 'D', 'es-ES')
FROM purchases

SELECT 'Date is: ' + FORMAT(CAST(Date AS DATE), 'dddd dd MM yyyy')
FROM purchases
```

Converting and formatting between number and string data types

```
SELECT Out, FORMAT(Out, '000.00'), CAST(Out as INT), STR(Out)
FROM purchases

SELECT Subtype, CAST(Subtype as CHAR(20)) + '.'
FROM purchases
```

Filtering data using the WHERE clause

```
SELECT * FROM purchases WHERE Out = 7.99  -- or >, <, >=, <=, and != or <>
SELECT * FROM purchases WHERE Out = 7.99 OR Out = 7.9
SELECT * FROM purchases WHERE Out IN (7.9, 7.99)

SELECT * FROM purchases WHERE Out >= 7.9 AND Out <= 15

SELECT * FROM purchases WHERE Out BETWEEN 7.9 AND 15

SELECT * FROM purchases WHERE Date = '25-Feb-27'

SELECT * FROM purchases WHERE CAST(Date as DATE) >= '2027-02-25'

SELECT * FROM purchases WHERE Subtype = 'Paper'
SELECT * FROM purchases WHERE Subtype = 'Ebook'
SELECT * FROM purchases WHERE LEFT(Subtype,5) = 'Ebook'
SELECT * FROM purchases WHERE Subtype LIKE 'Ebook%'
SELECT * FROM purchases WHERE Subtype LIKE '_book%'
SELECT * FROM purchases WHERE UPPER(Subtype) LIKE '%S%'
```

Grouping and Re-filtering data

```sql
SELECT PurchaseMethod, SUM(Out), COUNT(Out), MIN(Out), MAX(Out), AVG(Out)
FROM purchases
GROUP BY PurchaseMethod

SELECT PurchaseMethod, SUM(Out) as SumOut, COUNT(Out), MIN(Out), MAX(Out), AVG(Out)
FROM purchases
WHERE Out > 6
GROUP BY PurchaseMethod
HAVING Sum(Out) > 40

SELECT PurchaseMethod, Subtype, SUM(Out) AS SumOut, COUNT(*)
FROM purchases
GROUP BY PurchaseMethod, Subtype
```

Sorting the results and Using all 6 SQL clauses

```sql
SELECT PurchaseMethod, Subtype, Out
FROM purchases
ORDER BY Out

SELECT PurchaseMethod, Subtype, Out
FROM purchases
ORDER BY Out DESC

SELECT TOP 3 PurchaseMethod, Subtype, Out
FROM purchases
ORDER BY PurchaseMethod DESC, Subtype

SELECT PurchaseMethod, SUM(Out) as SumOut
```

```
FROM purchases
WHERE Out > 6
GROUP BY PurchaseMethod
HAVING SUM(Out) > 38
ORDER BY SumOut DESC
```

Identify and resolve duplicate data

```
SELECT DISTINCT PurchaseMethod
FROM purchases

SELECT PurchaseMethod
FROM purchases
GROUP BY PurchaseMethod

SELECT PurchaseMethod, Subtype, COUNT(*)
FROM purchases
GROUP BY PurchaseMethod, Subtype
HAVING COUNT(*)>1

SELECT DISTINCT Out
FROM purchases
```

Merging data

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

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

Joining data

```
SELECT Date, P.Subtype AS PurchasesSubtype, PurchaseMethod, Out, Category, PC.Subtype AS CategorySubtype
FROM purchases AS P
FULL JOIN purchasescategory AS PC -- or JOIN/INNER JOIN, LEFT JOIN, RIGHT JOIN
ON P.Subtype = PC.Subtype

SELECT Date, P.Subtype AS PurchasesSubtype, PurchaseMethod, Out, Category, PC.Subtype AS CategorySubtype
FROM purchasescategory AS PC
CROSS JOIN purchases AS P
```

Resolving missing data or null values

```
SELECT P.Subtype AS PurchasesSubtype, PC.Subtype AS CategorySubtype
     , COALESCE(P.Subtype, PC.Subtype)
FROM purchases AS P
FULL JOIN purchasescategory AS PC
ON P.Subtype = PC.Subtype
```

Conditional Functions

```
SELECT P.Subtype AS PurchasesSubtype, PC.Subtype AS CategorySubtype
     , CASE WHEN P.Subtype IS NOT NULL THEN P.Subtype
            WHEN PC.Subtype IS NOT NULL THEN PC.Subtype
```

```
            ELSE NULL END
     , CASE P.Subtype WHEN 'Misc' THEN 'Miscellaneous'
                      WHEN 'Other' THEN 'Miscellaneous' END
     , IIF(P.Subtype = 'Misc' OR P.Subtype = 'Other', 'Miscellaneous', NULL)
     , CHOOSE(3, 'A', 'B', 'C', 'D', 'E')
     , GREATEST(1, 2, 3)
     , LEAST(1, 2, 3)
FROM purchases AS P
FULL JOIN purchasescategory AS PC
ON P.Subtype = PC.Subtype
```

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

```
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]
```

Implementing a bridge table for a warehouse

```
SELECT Country
FROM tblActualSum
UNION
SELECT Country
FROM tblTargetSum

SELECT B.Country, A.TotalActual, T.TotalTarget
FROM tblBridge AS B
LEFT JOIN tblActualSum AS A
ON B.Country = A.Country
LEFT JOIN tblTargetSum AS T
```

```
ON B.Country = T.Country
ORDER BY B.Country
```

Creating a running total

```
SELECT Country, Location, Actual, SUM(Actual) OVER(PARTITION BY Country ORDER BY Location) as RunningTotal
FROM tblActual
--ROWNUMBER  PARTITIONBY ORDERBY
```

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')
```

Data Loading bottlenecks in SQL queries

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

Performance improvements in SQL queries

```
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'
```

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 ddddddd`	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.
```