ADVANCED DAX

for Business Intelligence

★★★★★ *With Power BI Expert* ***Aaron Parry***

MAVEN ANALYTICS

COURSE STRUCTURE

This is a **project-based course**, for students looking to build expert-level DAX skills through hands-on demos & assignments

Course resources include:

- **Downloadable PDF ebook** to serve as a helpful reference when you're offline or on the go (*or just need a refresher!*)
- **Quizzes** and **Assignments** to test and reinforce key concepts throughout the course, with detailed step-by-step solutions
- **Interactive**, **hands-on demos** to keep you engaged, with **downloadable project files** that you can use to follow along from home

COURSE OUTLINE

1	**Prerequisite Skills Review**	*Address the core DAX concepts you should already know, including evaluation context, filter flow, measures, etc.*
2	**Introducing the Course Project**	*Introduce the Maven Roasters course project, and build the data model that we'll analyze throughout the course*
3	**The DAX Engines**	*Understand the DAX formula and storage engines, data types, VertiPaq encoding & compression methods, etc.*
4	**Tips & Best Practices**	*Review helpful tips for formatting queries, adding comments, handling errors, and using DAX variables*
5	**Scalar Functions**	*Explore common scalar functions, including rounding, information, conversion, and logical functions*
6	**Advanced CALCULATE**	*Review CALCULATE modifiers, context transition, interactions with tables, etc.*

COURSE OUTLINE

7	**Table & Filter Functions**	*Create calculated tables, review common filter functions, explore ways to create new datasets, etc.*
8	**Calculated Table Joins**	*Create calculated joins between physical and virtual tables*
9	**Relationship Functions**	*Explore expanded tables, physical & virtual relationships, common relationship functions, etc.*
10	**Iterator Functions**	*Explore iterator cardinality, nested iterators, context transition, RANKX, calculation granularity, etc.*
11	**Advanced Time Intelligence**	*Build date tables with DAX, compare custom time periods, manage fiscal calendars, etc.*
12	**SNEAK PEEK: Performance Tuning**	*Introduce the performance analyzer, common optimization techniques, and performance tuning tools*

SETTING EXPECTATIONS

1 This course is for users who **already have basic DAX skills**

- *It is **strongly recommended** that you complete our **Up & Running with Power BI Desktop** course, or have a solid understanding of measures, calculated columns, evaluation context, and common DAX functions*

2 What you see on your screen **may not always match mine**

- *Power BI features are updated frequently (often monthly), so tools and interface options may change over time*
- *I'm using Windows 10 on a **PC/Windows** machine (note that Power BI Desktop is currently **unavailable for Mac**)*

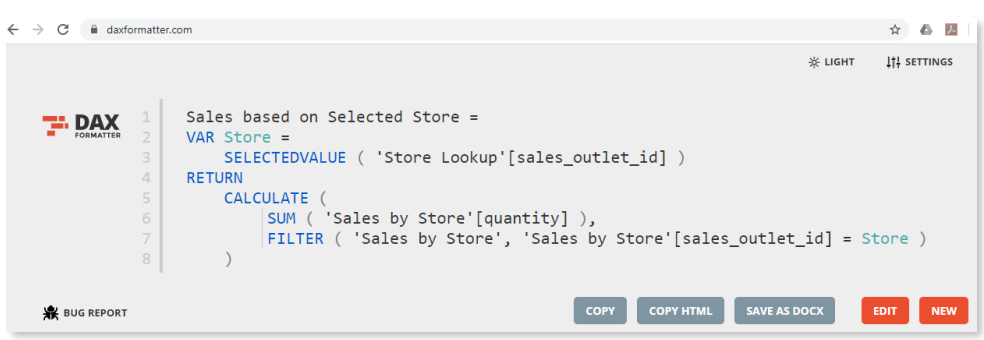
3 Our goal is to help you build **expert-level DAX skills**

- *DAX is simple to learn at a basic level, but advanced skills require a deep understanding of storage & calculation engines, table & filter functions, relationships, iterators, context transition, and more*

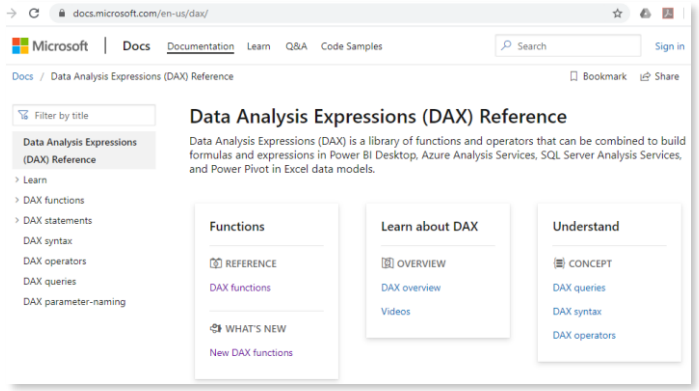
4 This course focuses on using DAX for **Business Intelligence**

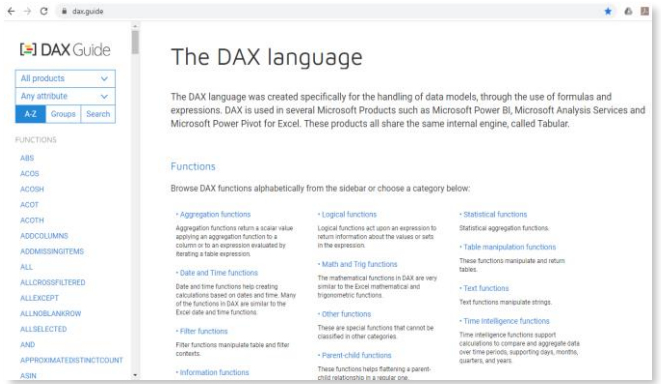
- *We'll take time to explore the inner-workings of the DAX engines, but our primary focus will be applying DAX to **real-world projects & case studies**; we're here to teach you how to become an **analyst**, not a **programmer***

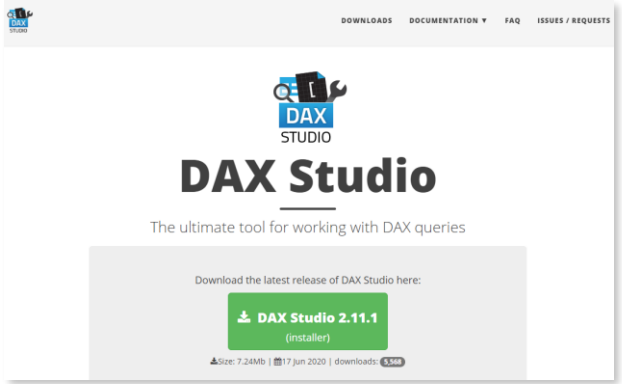
HELPFUL DAX RESOURCES



DAX Formatter *(daxformatter.com)* by sqlbi.com is a great tool for cleaning and formatting your DAX code, with options to customize based on regional settings or personal preferences







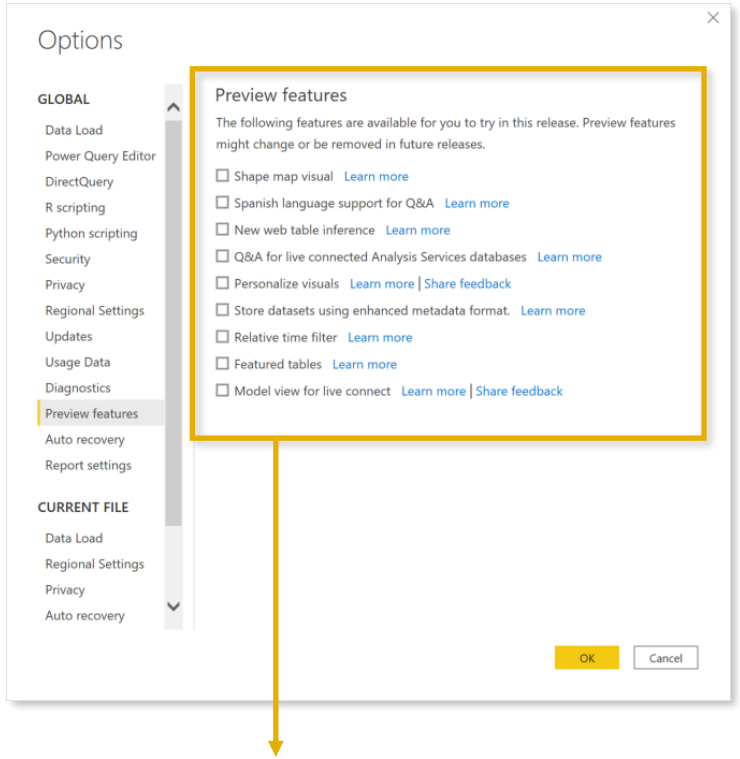
www.Docs.Microsoft.com/en-us/dax is the official Microsoft DAX reference guide, and a great resource for exploring DAX functions

DAX Guide *(dax.guide)* by sqlbi.com is a great resource to learn and explore DAX functions, category groups, product compatibility, and more

DAX Studio is an open source tool that allows you to write, execute and analyze DAX queries, as well as troubleshoot and optimize your code

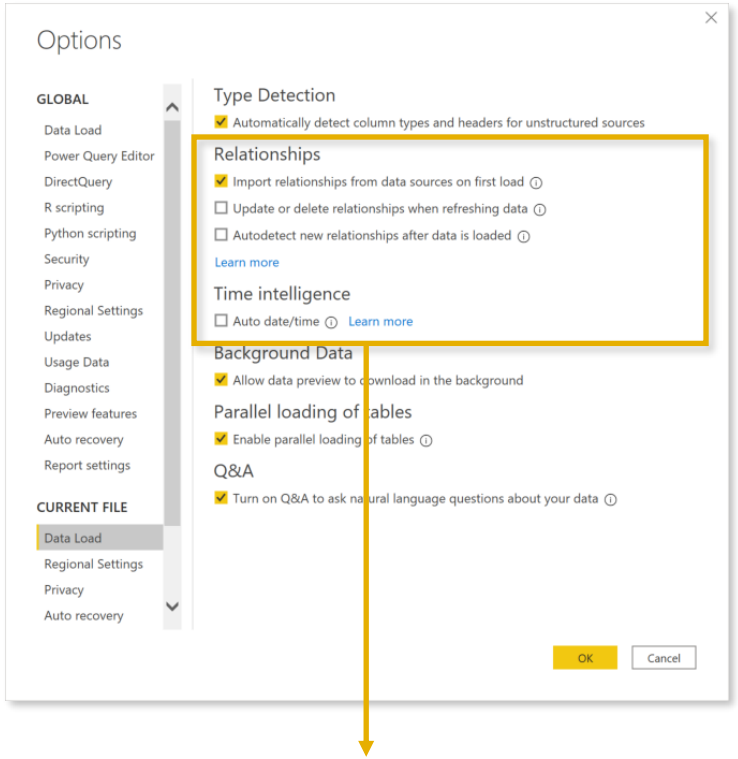
POWER BI OPTIONS & SETTINGS

PREVIEW FEATURES



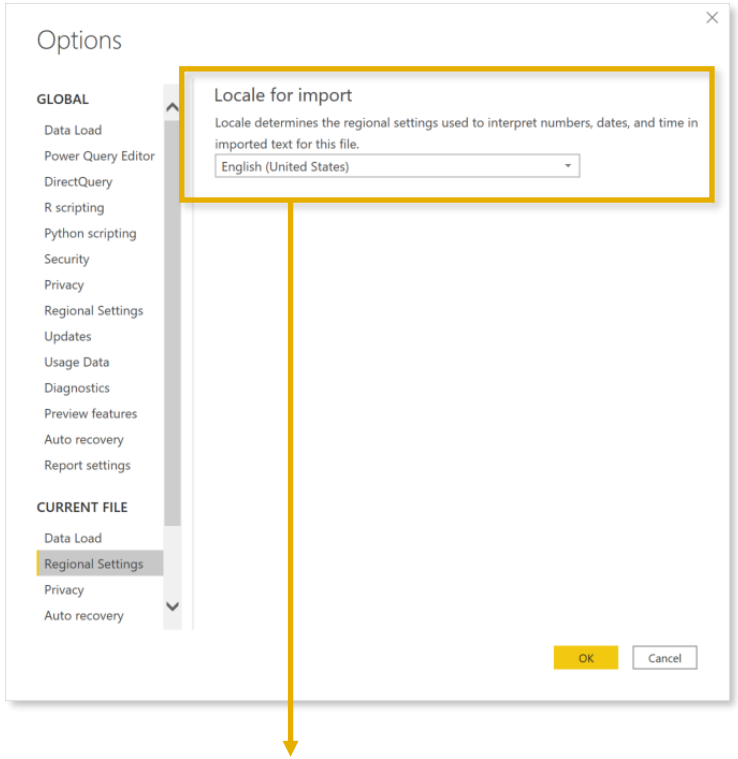
1) In the "**Preview Features**" tab, deselect any active features while you are taking this course

DATA LOAD



2) In the "**Data Load**" tab, deselect the "*Update Relationships*", "*Autodetect new relationships after data is loaded*" and "*Auto date/*time" options

REGIONAL SETTINGS



3) In the "**Regional Settings**" tab, make sure to use the "*English (United States)*" locale for import

NOTE: You may need to update setting in both the **Current File** and **Global** sections

PREREQUISITE SKILLS REVIEW

PREREQUISITE SKILLS REVIEW

This section reviews the core concepts you should already be familiar with, including **data modeling fundamentals** *(cardinality, filter flow, etc.)* and **basic DAX** *(operators, evaluation context, common measures, etc.)*

TOPICS WE'LL REVIEW:

- Data & Lookup Tables
- Primary & Foreign Keys
- Relationship Cardinality
- Filter Flow
- DAX Operators
- Common Functions
- Columns & Measures
- Evaluation Context

DATA TABLES VS. LOOKUP TABLES

Models generally contain two types of tables: **data** (or "*fact*") tables, and **lookup** (or "*dimension*") tables

- **Data tables** contain measurable *values* or *metrics* about the business *(quantity, revenue, pageviews, etc.)*
- **Lookup tables** provide descriptive *attributes* about each dimension in your model *(customers, products, etc.)*

date	product_id	quantity
1/1/1997	869	5
1/1/1997	1472	3
1/1/1997	76	4
1/1/1997	320	3
1/1/1997	4	4
1/1/1997	952	4
1/1/1997	1222	4
1/1/1997	517	4
1/1/1997	1359	4
1/1/1997	357	4
1/1/1997	1426	5
1/1/1997	190	4
1/1/1997	367	4
1/1/1997	250	5
1/1/1997	600	4
1/1/1997	702	5

*This **Data Table** contains "**quantity**" values, and connects to lookup tables via the "**date**" and "**product_id**" columns*

date	day_of_month	month	year	weekday	week_of_year	week_ending	month_name	quarter
1/1/1997	1	1	1997	Wednesday	1	1/5/1997	January	Q1
1/2/1997	2	1	1997	Thursday	1	1/5/1997	January	Q1
1/3/1997	3	1	1997	Friday	1	1/5/1997	January	Q1
1/4/1997	4	1	1997	Saturday	1	1/5/1997	January	Q1
1/5/1997	5	1	1997	Sunday	2	1/5/1997	January	Q1
1/6/1997	6	1	1997	Monday	2	1/12/1997	January	Q1

*This **Calendar Lookup** table provides additional attributes about each **date** (month, year, weekday, quarter, etc.)*

product_id	product_brand	product_name	product_sku	product_retail_price	product_cost	product_weigh
1	Washington	Washington Berry Juice	90748583674	2.85	0.94	8.39
2	Washington	Washington Mango Drink	96516502499	0.74	0.26	7.42
3	Washington	Washington Strawberry Drink	58427771925	0.83	0.4	13.1
4	Washington	Washington Cream Soda	64412155747	3.64	1.64	10.6
5	Washington	Washington Diet Soda	85561191439	2.19	0.77	6.66
6	Washington	Washington Cola	29804642796	1.15	0.37	15.8
7	Washington	Washington Diet Cola	20191444754	2.61	0.91	18
8	Washington	Washington Orange Juice	89770532250	2.59	0.8	8.97

*This **Product Lookup** table provides additional attributes about each **product** (brand, name, retail price, etc.)*

PRIMARY VS. FOREIGN KEYS

date	product_id	quantity
1/1/1997	869	5
1/1/1997	1472	3
1/1/1997	76	4
1/1/1997	320	3
1/1/1997	4	4
1/1/1997	952	4
1/1/1997	1222	4
1/1/1997	517	4
1/1/1997	1359	4
1/1/1997	357	4
1/1/1997	1426	5
1/1/1997	190	4
1/1/1997	367	4
1/1/1997	250	5
1/1/1997	600	4
1/1/1997	702	5

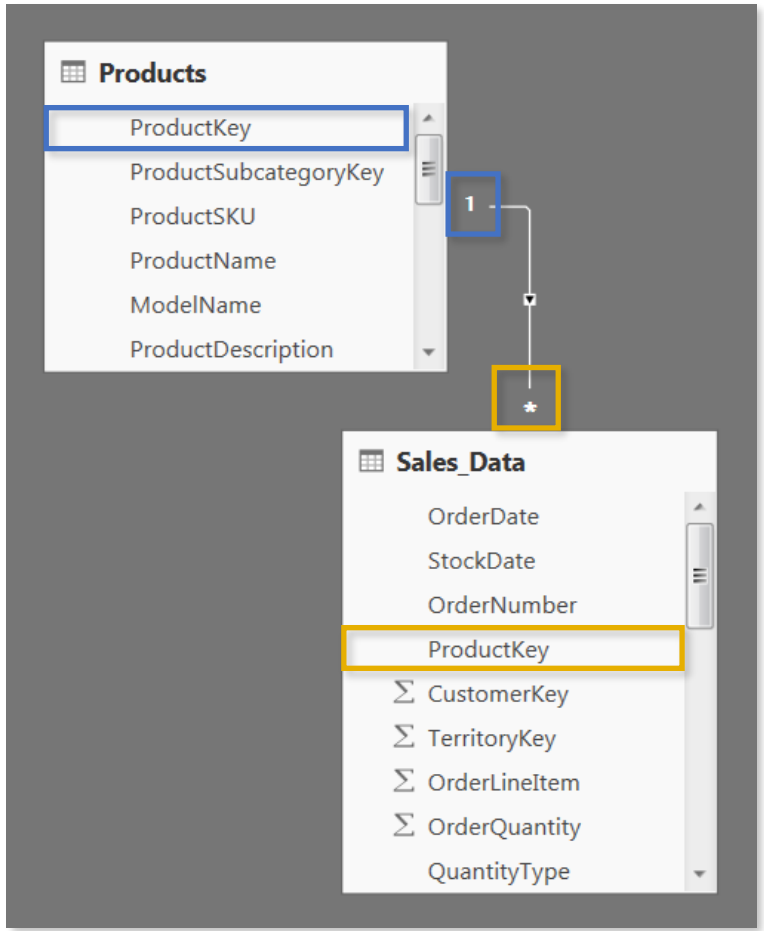
These columns are **foreign keys**; they contain *multiple* instances of each value, and are used to match the **primary keys** in related lookup tables

date	day_of_month	month	year	weekday	week_of_year	week_ending	month_name	quarter
1/1/1997	1	1	1997	Wednesday	1	1/5/1997	January	Q1
1/2/1997	2	1	1997	Thursday	1	1/5/1997	January	Q1
1/3/1997	3	1	1997	Friday	1	1/5/1997	January	Q1
1/4/1997	4	1	1997	Saturday	1	1/5/1997	January	Q1
1/5/1997	5	1	1997	Sunday	2	1/5/1997	January	Q1
1/6/1997	6	1	1997	Monday	2	1/12/1997	January	Q1

product_id	product_brand	product_name	product_sku	product_retail_price	product_cost	product_weight
1	Washington	Washington Berry Juice	90748583674	2.85	0.94	8.39
2	Washington	Washington Mango Drink	96516502499	0.74	0.26	7.42
3	Washington	Washington Strawberry Drink	58427771925	0.83	0.4	13.1
4	Washington	Washington Cream Soda	64412155747	3.64	1.64	10.6
5	Washington	Washington Diet Soda	85561191439	2.19	0.77	6.66
6	Washington	Washington Cola	29804642796	1.15	0.37	15.8
7	Washington	Washington Diet Cola	20191444754	2.61	0.91	18
8	Washington	Washington Orange Juice	89770532250	2.59	0.8	8.97

These columns are **primary keys**; they *uniquely* identify each row of a table, and match the **foreign keys** in related data tables

RELATIONSHIP CARDINALITY



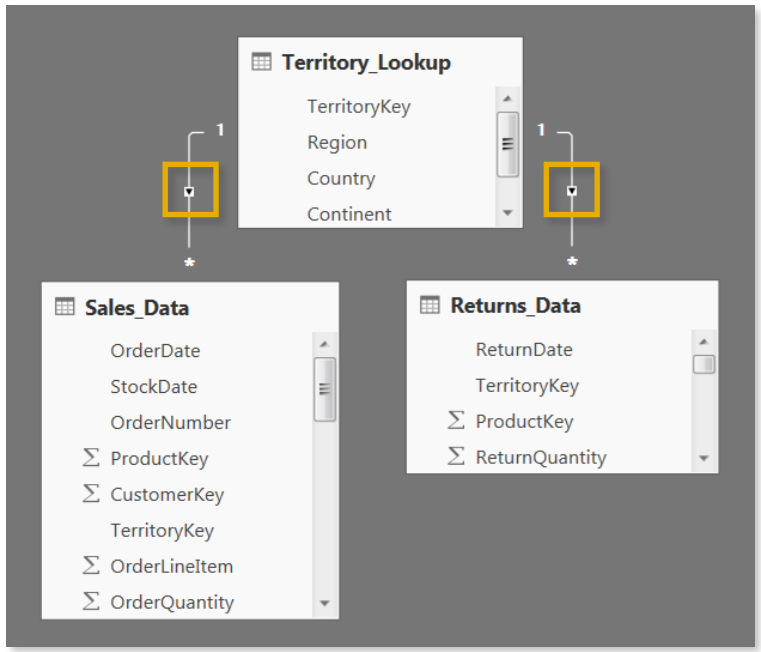
Cardinality refers to the *uniqueness of values* in a column

- For our purposes, all relationships in the data model should follow a "**one-to-many**" cardinality; **one** instance of each *primary key*, but potentially **many** instances of each *foreign key*

*In this case, there is only **ONE instance of each ProductKey** in the Products table (noted by the "**1**"), since each row contains **attributes of a single product** (Name, SKU, Description, Retail Price, etc)*

*There are **MANY instances of each ProductKey** in the Sales_Data table (noted by the asterisk *****), since there are **multiple sales associated with each product***

FILTER FLOW



Here we have two data tables (**Sales_Data** and **Returns_Data**), connected to **Territory_Lookup**

Note the filter directions (shown as arrows) in each relationship; by default, **these will point from the "one" side of the relationship** *(lookups)* **to the "many" side** *(data)*

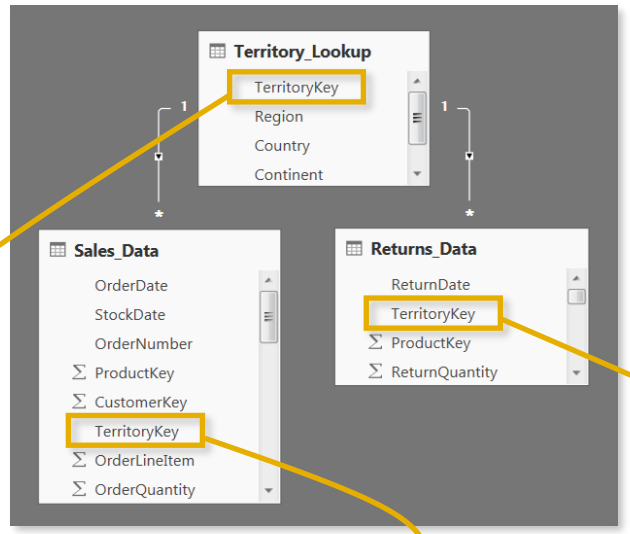
- When you filter a table, that filter context is passed along to all related "*downstream*" tables (following the direction of the arrow)
- Filters **cannot** flow "*upstream*" (against the direction of the arrow)

PRO TIP:

*Arrange your lookup tables **above** your data tables in your model as a visual reminder that filters flow "**downstream**"*

**In some cases filters may default to "two-way" depending on your Power BI Desktop settings*

FILTER FLOW (CONT.)



In this case, the only valid way filter both **Sales** and **Returns** data by Territory is to use the *TerritoryKey* field from the **Territory_Lookup** table, which is upstream and related to *both* data tables

- Filtering using *TerritoryKey* from the **Sales** table yields incorrect **Returns** values, since the filter context ***cannot flow upstream*** to either one of the other tables
- Similarly, filtering using *TerritoryKey* from the **Returns** table yields incorrect **Sales** data; in addition, ***only territories that registered returns are visible in the table*** (even though they registered sales)

TerritoryKey	OrderQuantity	ReturnQuantity
1	12,513	270
2	40	
3	30	
4	17,191	362
5	49	1
6	894	238
7	7,862	186
8	7,950	163
9	17,951	404
10	9,694	204
Total	**84,174**	**1,828**

1) *Filtering using TerritoryKey from the **Territory_Lookup** table*

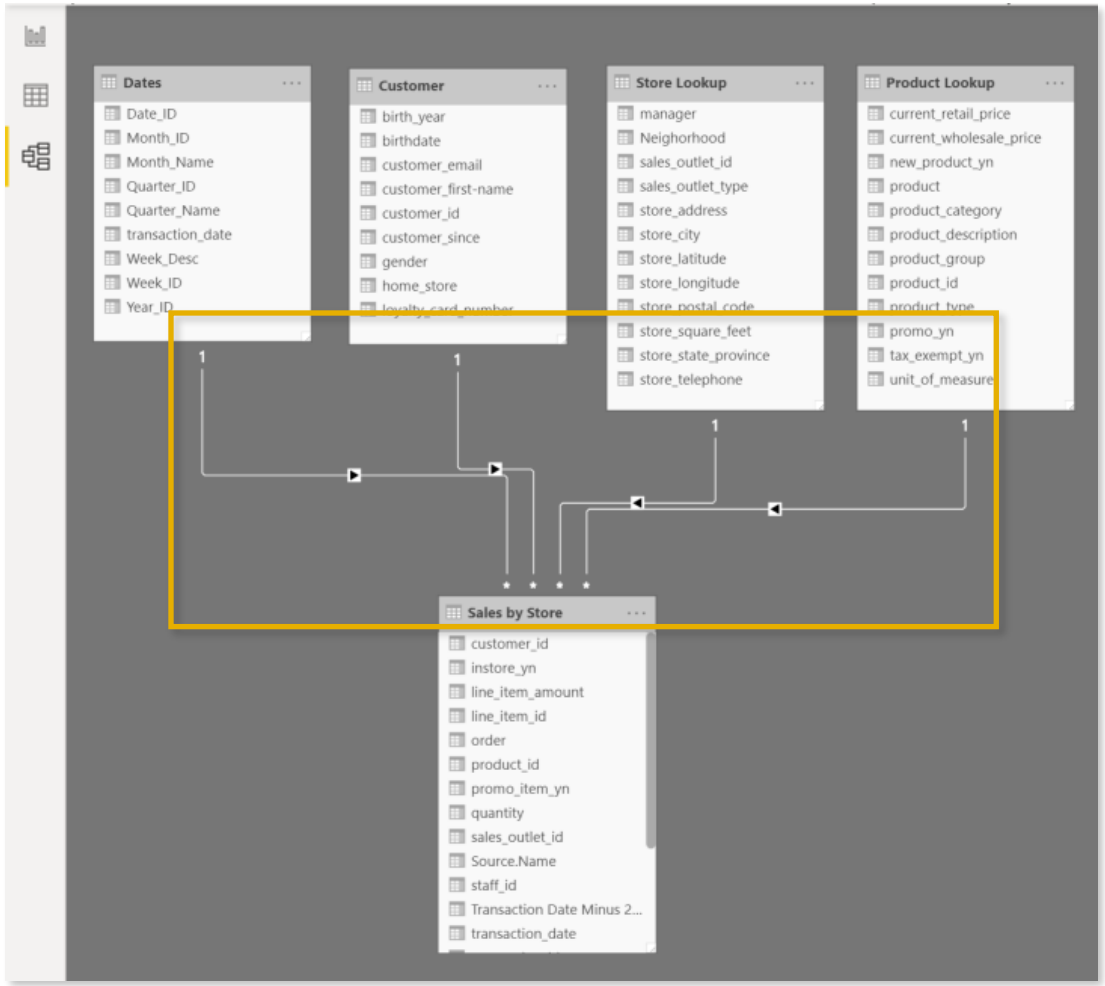
TerritoryKey	OrderQuantity	ReturnQuantity
1	12,513	1,828
2	40	1,828
3	30	1,828
4	17,191	1,828
5	49	1,828
6	894	1,828
7	7,862	1,828
8	7,950	1,828
9	17,951	1,828
10	9,694	1,828
Total	**84,174**	**1,828**

2) *Filtering using TerritoryKey from the **Sales_Data** table*

TerritoryKey	OrderQuantity	ReturnQuantity
1	84,174	270
4	84,174	362
5	84,174	1
6	174	238
7	4,174	186
8	84,174	163
9	84,174	404
10	84,174	204
Total	**84,174**	**1,828**

3) *Filtering using TerritoryKey from the **Returns_Data** table*

DATA MODEL BEST PRACTICES



A well-designed model is **critical** and ideally should:

- ✓ Use a star schema with **one-to-many** *(1:*)* relationships
- ✓ Contain relationships with **one-way** filters *(vs. bidirectional)*
- ✓ Contain tables that each serve a *specific* purpose, including **data** *(fact)* tables and **lookup** *(dim)* tables
- ✓ Only include the data you need for analysis *(no redundant or unnecessary records or fields)*
- ✓ Split out individual **date** and **time** components from **DateTime** fields

DAX OPERATORS

Arithmetic Operator	Meaning	Example
+	Addition	2 + 7
-	Subtraction	5 – 3
*	Multiplication	2 * 6
/	Division	4 / 2
^	Exponent	2 ^ 5

Comparison Operator	Meaning	Example
=	Equal to	[City]=“Boston”
>	Greater than	[Quantity]>10
<	Less than	[Quantity]<10
>=	Greater than or equal to	[Unit_Price]>=2.5
<=	Less than or equal to	[Unit_Price]<=2.5
<>	Not equal to	[Country]<>”Mexico”

Text/Logical Operator	Meaning	Example
&	Concatenates two values to produce one text string	[City] & “ “ & [State]
&&	Create an AND condition between two logical expressions	([State]=“MA”) && ([Quantity]>10)
|| (double pipe)	Create an OR condition between two logical expressions	([State]=“MA”) || ([State]=“CT”)
IN	Creates a logical OR condition based on a given list (using curly brackets)	‘Store Lookup’[State] IN { “MA”, “CT”, “NY” }

Head to **www.msdn.microsoft.com for more information about DAX syntax, operators, troubleshooting, etc.*

COMMON DAX FUNCTION CATEGORIES

MATH & STATS Functions

*Basic **aggregation** functions as well as "**iterators**" evaluated at the row-level*

Common Examples:

- SUM
- AVERAGE
- MAX/MIN
- DIVIDE
- COUNT/COUNTA
- COUNTROWS
- DISTINCTCOUNT

Iterator Functions:

- SUMX
- AVERAGEX
- MAXX/MINX
- RANKX
- COUNTX

LOGICAL Functions

*Functions for returning information about values in a given **conditional expression***

Common Examples:

- IF
- IFERROR
- AND
- OR
- NOT
- SWITCH
- TRUE
- FALSE

TEXT Functions

*Functions to manipulate **text strings** or **control formats** for dates, times or numbers*

Common Examples:

- CONCATENATE
- FORMAT
- LEFT/MID/RIGHT
- UPPER/LOWER
- PROPER
- LEN
- SEARCH/FIND
- REPLACE
- REPT
- SUBSTITUTE
- TRIM
- UNICHAR

FILTER Functions

***Lookup** functions based on related tables and **filtering** functions for dynamic calculations*

Common Examples:

- CALCULATE
- FILTER
- ALL
- ALLEXCEPT
- RELATED
- RELATEDTABLE
- DISTINCT
- VALUES
- EARLIER/EARLIEST
- HASONEVALUE
- HASONEFILTER
- ISFILTERED
- USERELATIONSHIP

DATE & TIME Functions

*Basic **date and time** functions as well as advanced **time intelligence** operations*

Common Examples:

- DATEDIFF
- YEARFRAC
- YEAR/MONTH/DAY
- HOUR/MINUTE/SECOND
- TODAY/NOW
- WEEKDAY/WEEKNUM

Time Intelligence Functions:

- DATESYTD
- DATESQTD
- DATESMTD
- DATEADD
- DATESINPERIOD

***Note:** This is NOT a comprehensive list (does not include trigonometry functions, parent/child functions, information functions, or other less common functions)*

CALCULATE

CALCULATE() — *Evaluates a given expression or formula under a set of defined filters*

=**CALCULATE**(Expression, *[Filter1], [Filter2],…*)

Name of an existing measure, or a DAX formula for a valid measure

Examples:

- *[Total Orders]*
- *SUM(Returns[ReturnQuantity])*

*List of simple Boolean (True/False) filter expressions (**Note:** these require simple, fixed values; you cannot create filters based on other measures)*

Examples:

- *Territory_Lookup[Country] = "**USA**"*
- *Calendar[Year] > **1998***

PRO TIP:

*CALCULATE works just like **SUMIF** or **COUNTIF** in Excel, except it can evaluate measures based on ANY sort of calculation (not just a sum, count, etc.); it may help to think of it like **"CALCULATEIF"***

CALCULATE (EXAMPLE)

```
Bike Returns = CALCULATE([Total Returns], Products[CategoryName] = "Bikes")
```

CategoryName	Total Returns	Bike Returns
Accessories	1,115	342
Bikes	342	342
Clothing	267	342
Components		342
Total	**1,724**	**342**

Here we've defined a new measure named "**Bike Returns**", which evaluates the "**Total Returns**" measure when the *CategoryName* in the **Products** table equals "**Bikes**"

*Wait, why do we see the **same repeating values** when we view a matrix with different categories on rows?*

*Shouldn't these cells have different filter contexts for **Accessories**, **Clothing**, **Components**, etc?*

HEY THIS IS IMPORTANT!

CALCULATE ***modifies*** and ***overrules*** any competing filter context!

In this example, the "Clothing" row has filter context of CategoryName = "**Clothing**" (*defined by the row label*) ***and*** CategoryName= "**Bikes**" (*defined by the CALCULATE function*)

Both cannot be true at the same time, so the "**Clothing**" filter is overwritten and the "**Bikes**" filter (from CALCULATE) takes priority

CALCULATE **CREATES NEW** FILTER CONTEXT

CALCULATE

Filters modified by CALCULATE

[**CategoryName**] = **"Bikes"**

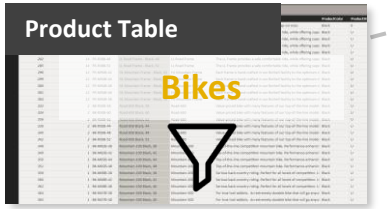
If the measure being evaluated contains a **CALCULATE** function, filter context is *overwritten* between **Step 1** & **Step 2**

STEP 1

Filter context is detected & applied

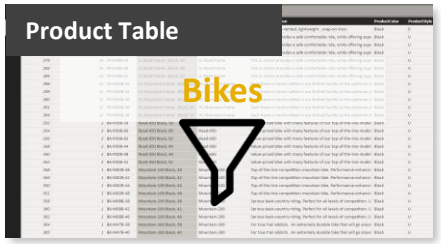
CategoryName	Total Returns	Bike Returns
Accessories	1,115	342
Bikes	342	342
Clothing	267	342
Components		342
Total	**1,724**	**342**

Products[**CategoryName**] = **"Accessories"**



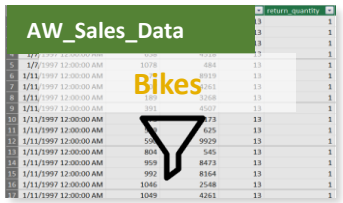
STEP 2

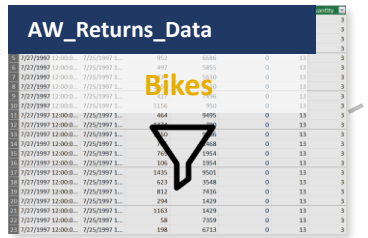
Filters flow "downstream" to all related tables



1 1

* *





STEP 3

Measure formula evaluates against the filtered table

```
Total Returns = COUNTROWS(AW_Returns_Data)
```

*Count of rows in the **AW_Returns_Data** table, filtered down to only rows where the product category is "**Bikes**"* = **342**

CALCULATED COLUMNS

Calculated columns allow you to add new, formula-based columns to tables

- No "A1-style" references; calculated columns refer to **entire tables** or **columns**
- Calculated columns generate values for each row, which are **visible within tables in the Data view**
- Calculated columns understand **row context**; they're great for defining properties based on information in each row, but generally useless for aggregation (*SUM, COUNT, etc.*)

HEY THIS IS IMPORTANT!

As a rule of thumb, use calculated columns when you want to "stamp" static, fixed values to each row in a table (*or use the Query Editor!*)

DO NOT use calculated columns for aggregation formulas, or to calculate fields for the "Values" area of a visualization *(use **measures** instead)*

PRO TIP:

*Calculated columns are typically used for **filtering** data, rather than creating numerical or aggregated values*

MEASURES

Measures are DAX formulas used to generate new calculated values

- Like calculated columns, measures reference **entire tables** or **columns** (*no A1-style or "grid" references*)
- *Unlike* calculated columns, **measure** values aren't visible within tables; they can only be "*seen*" within a visualization like a chart or matrix (*similar to a calculated field in an Excel pivot*)
- Measures are evaluated based on **filter context**, which means they recalculate when the fields or filters around them change (*like when new row or column labels are pulled into a matrix or when new filters are applied to a report*)

HEY THIS IS IMPORTANT!

As a rule of thumb, use measures (*vs. calculated columns*) when a single row can't give you the answer (*in other words, when you need to **aggregate***)

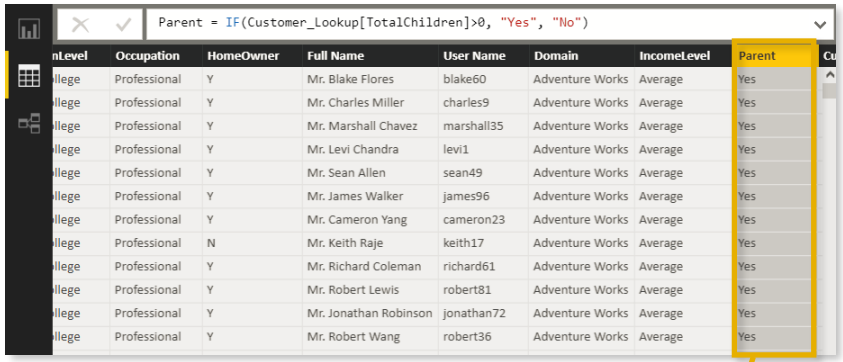
PRO TIP:

*Use measures to create **numerical, calculated values** that can be analyzed in the "**values**" field of a report visual*

RECAP: CALCULATED COLUMNS VS. MEASURES

CALCULATED COLUMNS

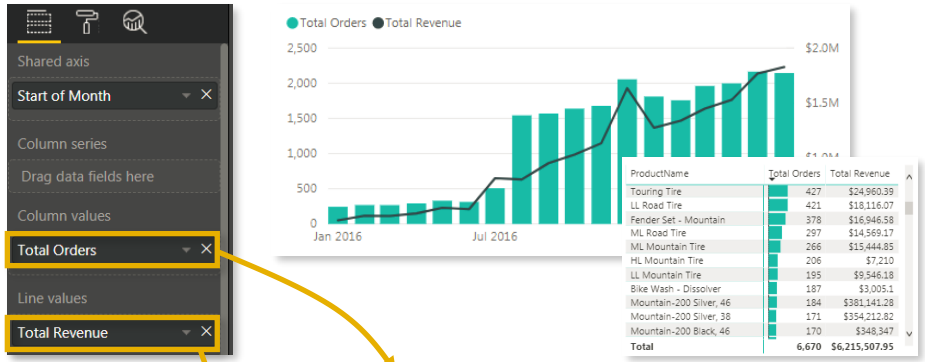
- Values are calculated based on information from each row of a table (has **row context**)
- Appends static values to each row in a table and stores them in the model (*which increases file size*)
- Recalculate on data source refresh or when changes are made to component columns
- Primarily used as **rows**, **columns**, **slicers** or **filters**



*Calculated columns "live" in **tables***

MEASURES

- Values are calculated based on information from any filters in the report (has **filter context**)
- Does not create new data in the tables themselves (*doesn't increase file size*)
- Recalculate in response to any change to filters within the report
- Almost *always* used within the **values** field of a visual



*Measures "live" in **visuals***

EVALUATION CONTEXT

Filter & **row context** tells DAX exactly how to evaluate measures and calculated columns in your data model (this is *key* for understanding advanced DAX topics)

FILTER CONTEXT

- Filter context **filters** the tables in your data model
- DAX creates filter context when dimensions are added to **rows**, **columns**, **slicers** & **filters** in a report
- **CALCULATE** can be used to systematically create or modify existing filter context
- Filter context always travels (propagates) from the **ONE** side to the **MANY** side of a table relationship

ROW CONTEXT

- Row context **iterates** through the rows in a table
- DAX creates row context when you add **calculated columns** to your data model
- **Iterator functions** *(SUMX, RANKX, etc.)* use row context to evaluate row-level calculations
- Row context doesn't automatically propagate through table relationships *(need to use **RELATED** or **RELATEDTABLE** functions)*

COURSE PROJECT

INTRODUCING THE COURSE PROJECT

THE SITUATION

You've just been hired as the lead Business Intelligence Analyst for **Maven Roasters***, a small-batch coffee chain based in New York City

THE BRIEF

All you've been given is a collection of raw csv files containing sales and inventory records, along with details about the company's **products**, **customers**, **stores** and **employees**.

Your goal is to use **Power BI** and **DAX** to answer key questions about the Maven Roasters business, including sales trends, inventory, top products, budgets and more.

THE OBJECTIVES

- Load the raw Maven Roasters data into Power BI
- Create table relationships to build a data model
- Use DAX to analyze key questions about the business

*This data is adapted from IBM #Cognos Analytics sample data and is for informational purposes only . These samples are provided "as is" without warranty of any kind. The example companies, organizations, products, domain names, email addresses, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, person, place, or event is intended or should be inferred.

THE MAVEN ROASTERS DATA MODEL

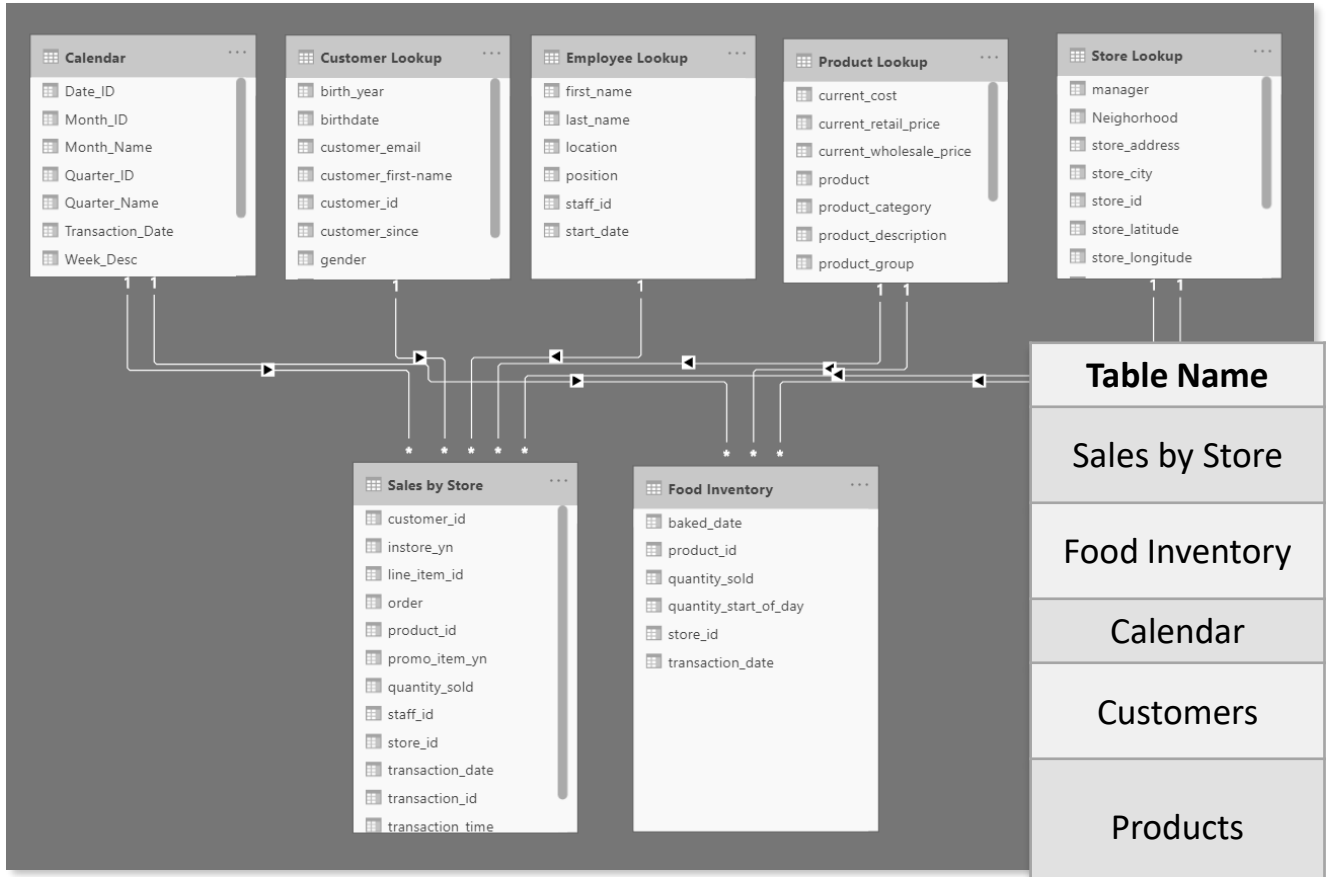


Table Name	Table Type	Definition
Sales by Store	Fact	*This table contains transaction data including date, store, employee, quantity sold, price, etc.*
Food Inventory	Fact	*This table contains details about daily pastry stock level, quantity used, and waste*
Calendar	Date	*Contains date, month, week, quarter and year details*
Customers	Lookup	*Customer ID, home store, name, email, loyalty details, birthdate, gender*
Products	Lookup	*Contains details for each product like type, group, category, unit of measure, wholesale price, retail price, tax exempt, discount and new product*
Stores	Lookup	*Details for each store location, address, phone number and store type*
Employees	Lookup	*Details for each staff member, their name, position, location and start date*

THE DAX ENGINES

THE DAX ENGINES

DAX is powered by two internal engines (**formula engine** & **storage engine**) which work together to compress & encode raw data and evaluate DAX queries

- Although these engines operate entirely behind the scenes, knowing how they work will help you build advanced skills by understanding exactly how DAX *thinks*

TOPICS WE'LL REVIEW:

- **Internal DAX Engines**
- **Query Evaluation**
- **Data & Storage Types**
- **Columnar Structures**
- **Compression & Encoding**
- **VertiPaq Relationships**

FORMULA & STORAGE ENGINES

- Internal DAX Engines
- Query Evaluation
- Data & Storage Types
- Columnar Structures
- Compression & Encoding
- VertiPaq Relationships

There are two distinct engines that work together to process every DAX query: the **Formula Engine** and the **Storage Engine**

FORMULA ENGINE

- Receives, interprets and executes all DAX requests
- Processes the DAX query then generates a list of logical steps called a **query plan**
- Works with the **datacache** sent back from the storage engine to evaluate the DAX query and return a result

STORAGE ENGINE

- Compresses and encodes raw data, and only communicates with the formula engine *(doesn't understand the DAX language)*
- Receives a query plan from Formula Engine, executes it, and returns a **datacache**
- **NOTE:** There are two types of storage engines, based on the type of connection you're using:

1. **VertiPaq** is used for data stored in-memory *(connected to Power BI via **import** mode)*
2. **DirectQuery** is used for data read directly from the source *(i.e. Azure, PostgreSQL, SAP)*

*We'll focus on the **VertiPaq** storage engine in this course*

QUERY EVALUATION IN DEPTH

- Internal DAX Engines
- **Query Evaluation**
- Data & Storage Types
- Columnar Structures
- Compression & Encoding
- VertiPaq Relationships

DAX QUERY:

```
Sales to Females =
CALCULATE(
    [Customer Sales],
    FILTER(
        'Customer Lookup',
        'Customer Lookup'[gender] = "F"
    )
)
```

$f(x)$ *DAX query is sent to the Formula Engine*

Formula Engine

*Formula engine interprets the query and sends a **query plan** to the storage engine*

Storage Engine

*Storage engine executes the plan and sends a **datacache** back to the formula engine*

Formula Engine

Formula engine runs the DAX query against the datacache, and evaluates the result

MEASURE OUTPUT:

Sales to Females:

$912,184

QUERY EVALUATION IN DEPTH

- Internal DAX Engines
- **Query Evaluation**
- Data & Storage Types
- Columnar Structures
- Compression & Encoding
- VertiPaq Relationships

DAX QUERY:

```
Sales to Females =
CALCULATE(
    [Total Sales],
    FILTER(
        Customer,
        Customer[gender] = "F"
    )
)
```

DAX here! Aaron would like me to evaluate this query

Formula Engine

Got it. Hey storage engine, can you grab some data for me?

Storage Engine

Sure! Here's the data you'll need to evaluate that query

Formula Engine

*Great! Hey Aaron, the result for that measure is **$912,184**!*

MEASURE OUTPUT:

Sales to Females:

$912,184

DATA & STORAGE TYPES

DAX uses **6 data types** to store values:

DAX Data Type	Power BI Data Type	Storage Type	Example
Integer	Whole Number	64-bit Value	*Max:* 9,223,372,036,854,775,807
Decimal	Decimal Number	Double-precision floating-point value	*64-bit precision*
Currency	Fixed Decimal Number	Fixed Decimal Number (Stored as Integer)	*317.9899*
DateTime	DateTime, Date, Time	Floating-point number	*1/1/2020 12:00p = 43830.50*
Boolean	True/False	True/False	*True/False*
String	Unicode String	16-bit characters	*"Maven Analytics" = "MAVEN ANALYTICS"*

HEY THIS IS IMPORTANT!

Data Types represent how values are ***stored*** by the DAX storage engine
Formatting represents how values ***appear*** to end users *(%, date, $, etc.)*

VERTIPAQ COLUMNAR DATA STRUCTURE

- Internal DAX Engines
- Query Evaluation
- Data & Storage Types
- **Columnar Structures**
- Compression & Encoding
- VertiPaq Relationships

VertiPaq uses a **columnar data structure**, which stores data as individual columns *(rather than rows or full tables)* to quickly and efficiently evaluate DAX queries

product_id	product_group	product_category	wholesale_price
9	Whole Bean/Teas	Coffee beans	$18.00
17	Whole Bean/Teas	Loose Tea	$7.60
21	Whole Bean/Teas	Packaged Chocolate	$10.66
57	Beverages	Tea	$0.78
61	Beverages	Drinking Chocolate	$3.56
65	Add-ons	Flavours	$0.04
79	Food	Bakery	$2.44
83	Merchandise	Branded	$4.48
88	Beverages	Coffee	$0.42

VERTIPAQ COMPRESSION & ENCODING

The goal of **compression** and **encoding** is to reduce the amount of memory needed to evaluate a DAX query.

Based on a sample of data, one (or more) of the following methods will be used:

1. **Value Encoding**
   - Mathematical process used to reduce the number of bits needed to store **integer** values
2. **Hash Encoding** *(aka Dictionary Encoding)*
   - Identifies the distinct **string** values and creates a new table with indexes
3. **Run Length Encoding** *(RLE)*
   - Reduces the size of a dataset by identifying repeated values found in adjacent rows

HEY THIS IS IMPORTANT!

The actual storage algorithms are proprietary, so not all details are available ☹

VALUE ENCODING

Value Encoding uses a mathematical process to determine relationships between the values in a column, and convert them into smaller values for storage

- Value encoding only works for integer values *(including currency)*, and cannot be applied to strings or floating-point values

This is what you see

City ID
10014
10106
10215
10007
10002
10021
10036

- *Max = **10215***
- ***14 bits** needed*

This is how the data is stored

City ID
14
106
215
7
2
21
36

- *Max = **215***
- ***9 bits** needed (**36**% reduction!)*

HASH ENCODING

Hash Encoding builds a "dictionary" of distinct items in a column, assigns a unique integer value *(index)* to each item, and stores the data using the index values rather than the full text strings

- With hash encoding, storage requirements are defined by the number of unique items in the column *(cardinality),* **NOT** by the length of the string values themselves

This is what you see

Product
Coffee Beans
Tea
Bakery
Bakery
Coffee Beans
Coffee Beans
Bakery
Tea

This is how the data is stored

Index	Row
0	1
1	2
2	3
2	4
0	5
0	6
2	7
1	8

Product	Index
Coffee Beans	0
Tea	1
Bakery	2

*Encoding **dictionary***

RUN LENGTH ENCODING

Run Length Encoding (RLE) reduces the size of a column by replacing duplicate rows with a table containing each distinct value and the count of instances

- ***NOTE:*** *RLE only works when the same value is repeated in consecutive rows, but the VertiPaq engine automatically sorts data on import and refresh to find the optimal compression*

This is what you see

City Zip	
59716	
59716	***3*** *instances*
59716	
02215	
02215	***2*** *instances*
05672	
05672	
05672	***4*** *instances*
05672	
05672	

This is how the data is stored

City Zip	Count
59716	3
02215	2
05672	4

One row *per distinct value*

RLE + HASH ENCODING

Columns can have both **Run Length** and *either* **Hash (Dictionary)** or **Value** encoding

- Compression type is determined by cardinality, number of repeat values, row count, and data type

City Zip
59716
59716
05672
05672
05672
05672
05672
90210
90210
90210
90210
83353

City Index	Zip
0	59716
1	05672
2	90210
3	83353

Hash Encoding

City Index	Row
0	1
0	2
1	3
1	4
1	5
1	6
1	7
2	8
2	9
2	10
2	11
3	12

City Index	Count
0	2
1	5
2	4
3	1

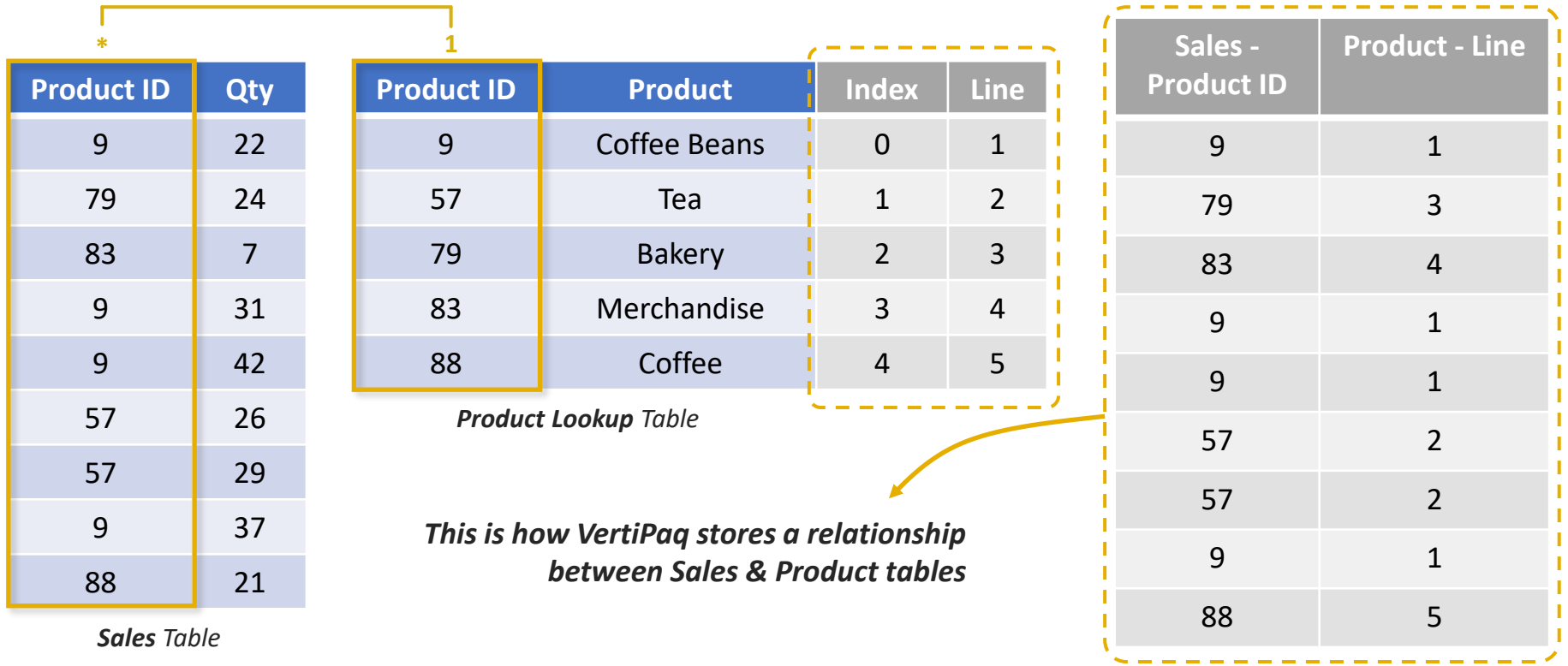
RLE Encoding

This is how the "City Zip" data is encoded and stored by VertiPaq

VERTIPAQ RELATIONSHIPS

VertiPaq has a special way of **mapping relationships** between columns in your data model, which allows it to evaluate complex, multi-column queries

- ***NOTE:*** *This is NOT the same as creating table relationships in your data model; this is essentially a blueprint that VertiPaq uses to map pairs of primary & foreign keys across related tables*



VERTIPAQ RELATIONSHIPS

- Internal DAX Engines
- Query Evaluation
- Data & Storage Types
- Columnar Structures
- Compression & Encoding
- **VertiPaq Relationships**

```
1 Coffee Bean Sales =
2 CALCULATE(
3     [Quantity Sold],
4     FILTER(
5         'Product Lookup',
6         'Product Lookup'[product] = "Coffee Beans"
7     )
8 )
```

STEP 1: Search for *"Coffee Beans"* in Product dictionary to find line number from product lookup

Product	Index	Line
Coffee Beans	0	1
Tea	1	2
Bakery	2	3
Merchandise	3	4
Coffee	4	5

STEP 2: Use **relationship** to find all rows where product line = 1

Sales - Product ID	Product - Line
9	1
79	3
83	4
9	1
9	1
57	2
57	2
9	1
88	5

STEP 3: Return a **datacache** containing a filtered sales table and sends to formula engine

Product ID	Qty
9	22
9	31
9	42
9	37

STEP 4: Formula engine evaluates the **[Quantity Sold]** measure against datacache

[Quantity Sold] = **132**

SUMMARY

DAX uses two engines: the **Formula Engine** & **Storage Engine**

- *The Formula Engine interprets & evaluates DAX, and the Storage Engine compresses & encodes data*

There are two types of storage engines: **VertiPaq** & **DirectQuery**

- ***VertiPaq** is used for in-memory storage and **DirectQuery** is used for direct connections to external sources; both create **datacaches** and send them to the formula engine for DAX query evaluation*

VertiPaq stores data using a **columnar database setup**

- *Data is stored in individual columns that can be accessed quickly, but queries that call multiple columns may require more complex logic to produce a datacache*

Raw data is **compressed** & **encoded** to optimize processing

- *Data can be compressed using **Value**, **Hash** (Dictionary), or **Run Length** encoding (RLE), based on cardinality, repeat values, row count, and data type*

TIPS & BEST PRACTICES

DAX TIPS & BEST PRACTICES

In this section we'll cover some common techniques and best practices for working efficiently with DAX, including **shortcuts**, **comments**, **measure tables**, and **variables**

TOPICS WE'LL COVER:

- **Shortcuts**
- **Formatting**
- **Evaluation Order**
- **Comments**
- **Measure Tables**
- **Error Handling**
- **Variables**

COMMON USE CASES:

- *Saving time with keyboard shortcuts*
- *Adding comments to help others understand exactly what each line of your DAX code is doing*
- *Using error checking functions for testing & QA*
- *Adding variables to troubleshoot code, enhance readability and improve DAX performance*

DAX SHORTCUTS

Action	Keyboard Shortcut
Insert Line Below	Shift + Enter
Insert Line Above	Control + Shift + Enter
Select Current Line	Control + I
Cut Current Line	Control + X
Paste Cut Line	Control + V
Duplicate Current Row	Shift + Alt + ↓↑
Go to Line	Control + G
Enlarge Code Type	Control + Mouse Scroll

PRO TIP:

*If you use a **square bracket** "[" when you call a measure, you'll avoid having to filter through function names that also start with the same letter(s) as your measure name*

*For more keyboard shortcuts visit: https://docs.microsoft.com/en-us/power-bi/transform-model/desktop-formula-editor

FORMATTING BEST PRACTICES

- Shortcuts
- Formatting
- Evaluation Order
- Comments
- Measure Tables
- Error Handling
- Variables

Formatting is **KEY** for readability. Can you tell what this code does in **10 seconds**?

Time's Up!

```
Sales for Selected Store = VAR Store = SELECTEDVALUE('Store Lookup'
[store_id]) RETURN CALCULATE(SUM('Sales by Store'[Quantity_Sold]) *
1.05,FILTER('Sales by Store','Sales by Store'[Store_ID] = Store  ))
```

```
Sales for Selected Store =
VAR Store =
SELECTEDVALUE('Store Lookup'[store_id]) -- Define column for single value

RETURN
CALCULATE(                              --Changes the filter context of
    SUM(                                --the quantity sold multiplied by 5%
        'Sales by Store'[Quantity_Sold]) * 1.05,
        FILTER(                         -- based on a filtered table
          'Sales by Store',
          'Sales by Store'[Store_ID] = Store  -- where selected store has one value
        )
)
```

PRO TIP

*Use **shift** + **enter** to split out and indent each component of your DAX formulas to make them more human readable (**TIP:** try using **daxformatter.com** to quickly format your code)*

EVALUATION ORDER

Evaluation order is the process by which DAX evaluates the parameters in a function

- **Individual functions** typically evaluate from **left-to-right**, starting with the first parameter *(followed by the second, third, etc.)*
- **Nested functions** evaluate from the **inside-out**, starting with the innermost function and working outward from there

Non-nested:

```
          1          2              3
=IF(LogicalTest, ResultIfTrue, [ResultIfFalse])
```

Nested:

```
=SUMX(
    FILTER(
        FILTER( 'Table',
            RELATED( 'Table'[Column]), 1
        RELATED( 'Table'[Column]), 2
    'Table'[Column]) 3
```

NOTE: The **CALCULATE** function evaluates using its own unique set of rules *(more on this later!)*

EVALUATION ORDER (EXAMPLE)

Non-Nested

```
1  Customer Sales =
2  SUMX(
3      'Sales by Store',                                                    -- 1. DAX returns the Sales by Store table
4      'Sales by Store'[quantity_sold] * 'Sales by Store'[unit_price]  -- 2. and then evaluates the expression
5  )
```

Nested

```
1  Store 3 Sales of Whole Bean/Teas (SUMX) =
2  SUMX(               -- 3. applies filters to Sales by Store to compute SUM of Quantity * Price
3      FILTER(         -- 2. then, DAX evaluates the next table output of FILTER
4          FILTER(     -- 1. DAX first evaluates the innermost table output of FILTER,
5              'Sales by Store',
6              RELATED(
7                  'Store Lookup'[sales_outlet_id]) = 3                -- 1
8          ),
9          RELATED(
10             'Product Lookup'[product_group]) = "Whole Bean/Teas"     -- 2
11     ),
12     'Sales by Store'[Quantity_Sold] * 'Sales by Store'[Unit_Price]  -- 3
13 )
```

COMMENTING YOUR CODE

Comments can help other users interpret your code, and can be particularly helpful for complex queries with multiple lines, nested functions, etc.

Comment Type	Marker
Single Line Comment	-- or //
Multi Line Comment	/* … */

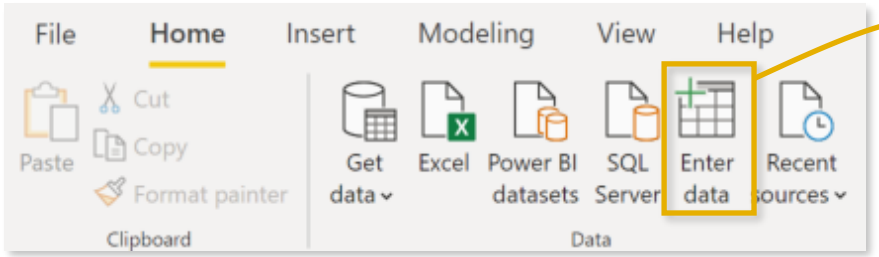
```
1  Sales based on Selected Store =
2  VAR Store =
3  SELECTEDVALUE('Store Lookup'[sales_outlet_id]) -- This is an example of a single line comment
4
5  RETURN
6  CALCULATE(
7      SUM('Sales by Store'[quantity]),
8      FILTER('Sales by Store', 'Sales by Store'[sales_outlet_id] = Store)  /*Or, you can explain how you're using
9      filter to return a table where the store id is based on a selected value with a multiline comment */
10 )
```

PRO TIP

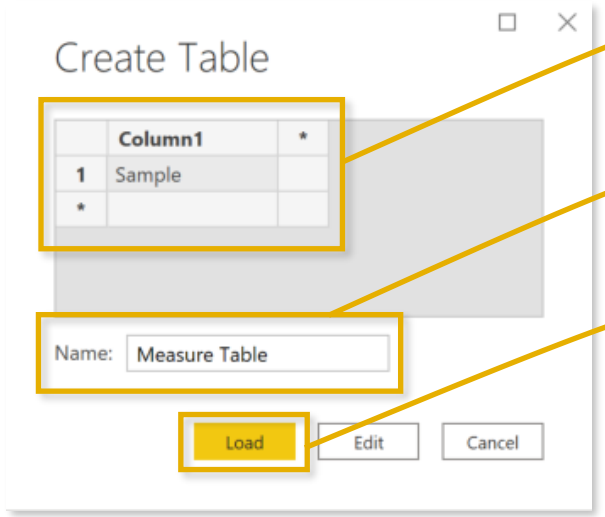
*Avoid putting comments **at the end of your DAX query** (below the last closing parenthesis), as they can be missed or omitted by users and formatting tools*

PRO TIP: DEDICATED MEASURE TABLES

Creating a **separate table** to contain your DAX measures is a great way to stay organized *(you can even group your measures into folders within the table!)*



Enter Data *to create a new table*



*Add a **placeholder value** in the first row of the table (can be anything you want)*

Name *the table (i.e. "**Measure Table**")*

Load *the table to your data model*

ERROR HANDLING

Error handling functions can be used to help identify missing data, and can be particularly useful for quality assurance and testing

IFERROR()	*Returns a value if the first expression is an error and the value of the expression itself otherwise*	=**IFERROR**(Value,ValueIfError)
ISBLANK()	*Checks to see if a value is blank, returns True or False*	=**ISBLANK**(Value)

```
1  Error Checking Example =
2  IFERROR(
3      1/0,
4      BLANK()
5  )
```

```
1  Customer Sales LY (ISBLANK) =
2  IF(
3      ISBLANK(
4          [Customer Sales (Last Year)]
5      ),
6      "No Sales",
7      [Customer Sales (Last Year)]
8  )
```

PRO TIP:

*VertiPaq can't optimize **IFERROR** and **ISBLANK** functions, so avoid using them in your operational code (a better use is for temporary QA & testing)*

DAX VARIABLES

Variables ***(VAR)*** are DAX expressions which can be reused multiple times within a query, and are commonly used for **two key purposes**:

- **Readability**: Variables make complex code more human readable
- **Performance**: Variables are only evaluated *once* no matter how often they are used within a query

What they **CAN** do:

- **Simplify** and **streamline** DAX code
- **Improve efficiency** by eliminating redundant expressions
- Evaluate **in the order they're written** *(variables can only reference previously declared variables within the query)*
- Store either **table** or **scalar** values

What they **CANNOT** do:

- Start with a **number**
- Include **spaces** or **special characters** *(except underscores "_")*
- **Share the name** of another table in the model
- Be accessed **outside the query** in which they are declared
- Contain only **certain keywords** reserved for DAX *(SUM, Date, CALCULATE, etc.)*

CREATING & USING VARIABLES

- Shortcuts
- Formatting
- Evaluation Order
- Comments
- Measure Tables
- Error Handling
- Variables

DAX queries which use variables must include two key components: the **declaration expression** and the **return expression**:

VAR <name> = <expression> **RETURN** <result_expression>

```
1  Orders by Females =
2  VAR TotalOrders_Female =
3  CALCULATE(
4      SUM(
5       'Sales by Store'[quantity_sold]
6      ),
7      FILTER(
8          'Customer Lookup',
9          'Customer Lookup'[gender] = "F"
10      )
11 )
12 RETURN
13 TotalOrders_Female
```

The **declaration expression (VAR)** is where you declare a new variable, assign a name, and write an expression to define the variable

The **return expression (RETURN)** is where you evaluate the rest of your DAX query, and reference any previously declared variables

VARIABLE EVALUATION ORDER

Variables are “locked in” as soon as the DAX engine reads them; this means you cannot modify how a variable is defined later in your query *(i.e. through a CALCULATE function)*

*Total **quantity sold** to Female customers* = **283,427**

```
1  Orders by Females - Wrong (VAR Evaluation Order) =
2  VAR TotalOrders =
3  SUM(
4      'Sales by Store'[quantity_sold]
5  )
6  RETURN
7  CALCULATE(
8      TotalOrders,
9      FILTER(
10         'Customer Lookup',
11         'Customer Lookup'[gender] = "F"
12     )
13 )
```

= **1,305,637**

***Variable** can’t be modified after it’s been defined*

```
1  Orders by Females - Right (VAR Evaluation Order) =
2  VAR TotalOrders_Female =
3  CALCULATE(
4      SUM(
5       'Sales by Store'[quantity_sold]
6      ),
7      FILTER(
8          'Customer Lookup',
9          'Customer Lookup'[gender] = "F"
10     )
11 )
12 RETURN
13 TotalOrders_Female
```

= **283,427**

PRO TIP: VARIABLES TO TEST & DEBUG DAX

- Shortcuts
- Formatting
- Evaluation Order
- Comments
- Measure Tables
- Error Handling
- **Variables**

Variables can be a helpful tool for **testing** or **debugging** your DAX code

```
% Quantity Sold to Females (VAR) =
VAR TotalFemaleOrders =
CALCULATE(
    SUM('Sales by Store'[quantity_sold]),
    FILTER(
        'Customer Lookup',
        'Customer Lookup'[gender] = "F"
    )
)
VAR QuantitySold =
SUM(
    'Sales by Store'[quantity_sold]
)
VAR Ratio =
DIVIDE(
    TotalFemaleOrders,
    QuantitySold,
    "-"
)

RETURN
Ratio
```

In this example we're able to:

- Use variables to define individual components of a larger, more complex measure
- Use the RETURN expression to quickly check the output of each variable
- Identify which specific component is the root cause in the case of an error

SCALAR FUNCTIONS

SCALAR FUNCTIONS

Scalar functions return a **single value**, rather than a column or table; common examples include aggregation, conversion, rounding, and logical functions

TOPICS WE'LL COVER:

- **Aggregation Functions**
- **Rounding Functions**
- **Information Functions**
- **Conversion Functions**
- **Logical Functions**

COMMON USE CASES:

- *Aggregating a column of values into a single number (i.e. average customer age, maximum product price, sum of revenue, count of orders, etc.)*
- *Converting fields into desired formats (i.e. text to dates, integers to currency, etc.)*
- *Evaluating logical tests and returning values for TRUE and FALSE responses*

COMMON SCALAR FUNCTIONS

AGGREGATION Functions

*Functions that can be used to **dynamically aggregate** values within a column*

Common Examples:

- SUM
- AVERAGE
- MAX
- MIN
- COUNT
- COUNTA
- DISTINCTCOUNT
- PRODUCT
- ITERATOR (“X”) FUNCTIONS

ROUNDING Functions

*Functions that can be used to **round values** to different levels of precision*

Common Examples:

- FLOOR
- TRUNC
- ROUNDDOWN
- MROUND
- ROUND
- CEILING
- ISO.CEILING
- ROUNDUP
- INT
- FIXED

INFORMATION Functions

*Functions that can be used to analyze the **data type** or output of an expression*

Common Examples:

- ISBLANK
- ISERROR
- ISLOGICAL
- ISNONTEXT
- ISNUMBER
- ISTEXT

CONVERSION Functions

*Functions that are used to force a specific **data type conversion***

Common Examples:

- CURRENCY
- INT
- FORMAT
- DATE
- TIME
- DATEVALUE
- VALUE

LOGICAL Functions

*Functions for returning information about values in a **conditional expression***

Common Examples:

- IF
- AND
- OR
- NOT
- TRUE/FALSE
- SWITCH
- COALESCE

AGGREGATION FUNCTIONS

- Aggregation Functions
- Rounding Functions
- Information Functions
- Conversion Functions
- Logical Functions

SUM()	*Evaluates the sum of a column*	=SUM(ColumnName)
AVERAGE()	*Returns the average (arithmetic mean) of all the numbers in a column*	=AVERAGE(ColumnName)
MAX()	*Returns the largest value in a column or between two scalar expressions*	=MAX(ColumnName) *or* =MAX(Scalar1, *[Scalar2]*)
MIN()	*Returns the smallest value in a column or between two scalar expressions*	=MIN(ColumnName) *or* =MIN(Scalar1, *[Scalar2]*)
COUNT()	*Counts the number of cells in a column that contain numbers*	=COUNT(ColumnName)
DISTINCTCOUNT()	*Counts the number of distinct or unique values in a column*	=DISTINCTCOUNT(ColumnName)
COUNTROWS()	*Counts the number of rows in the specified table, or a table defined by an expression*	=COUNTROWS(Table)

PRO TIP:

*For large datasets (1M+ rows) using **COUNTROWS** & **VALUES** may put less strain on the DAX engines than **DISTINCTCOUNT***

```
1  Total Employees =
2  COUNTROWS(
3      VALUES(
4          'Employee Lookup'
```

PRO TIP: SUM VS. SUMX

There are several cases where DAX evaluates a basic query using a more complex method behind the scenes. The simplified query is often called *"syntax sugar"*

- For example, **SUM** is read internally as **SUMX**

How it's written:

```
1 Total Sales =
2 SUM('Sales by Store'[quantity_sold])
```

How it's interpreted by DAX:

```
1 Total Sales =
2 SUMX(
3     'Sales by Store',
4     'Sales by Store'[quantity_sold])
```

HEY THIS IS IMPORTANT!

This is how *all* aggregation functions are processed internally by DAX *(SUM, AVERAGE, MAX, etc.)*

ROUNDING FUNCTIONS

- Aggregation Functions
- **Rounding Functions**
- Information Functions
- Conversion Functions
- Logical Functions

INT()	*Rounds a number down to the nearest integer*	=**INT**(Number)
ROUND()	*Rounds a number to a specific number of digits*	=**ROUND**(Number, NumberOfDigits)
ROUNDUP()	*Rounds a number up, away from zero*	=**ROUNDUP**(Number, NumberOfDigits)
ROUNDDOWN()	*Rounds a number down, toward zero*	=**ROUNDDOWN**(Number NumberOfDigits)
MROUND()	*Rounds a number to the desired multiple*	=**MROUND**(Number, Multiple)
TRUNC()	*Truncates a number to an integer by removing the decimal part of the number*	=**TRUNC**(Number, *[NumberOfDigits]*)
FIXED()	*Rounds number down to specified number of decimals and returns result as text*	=**FIXED**(Number, *[Decimals], [No.Commas]*)
CEILING()	*Rounds a number up, to the nearest integer or nearest unit of significance*	=**CEILING**(Number, Significance)
FLOOR()	*Rounds a number down, toward zero, to the nearest multiple of significance*	=**FLOOR**(Number, Significance)

ROUNDING FUNCTION EXAMPLES

- Aggregation Functions
- **Rounding Functions**
- Information Functions
- Conversion Functions
- Logical Functions

Decimal Value: **3.12438**

INT *(3.12438) =* ***3***

Useful as a component in DAX statements, like calendar tables

ROUND *(3.12438, 2) =* ***3.12***

ROUNDUP *(3.12438, 2) =* ***3.13***

ROUNDDOWN *(3.12438, 2) =* ***3.12***

Useful to specify the precision of a number, like customer age

FIXED *(3.12438, 2) =* ***3.12***

Useful when you want to convert a number to text

Time value: **9:34:14 AM**

MROUND *(9:34:14, ”0:15”) =* ***9:30:00 AM***

Rounds the minute component up or down based on the multiple

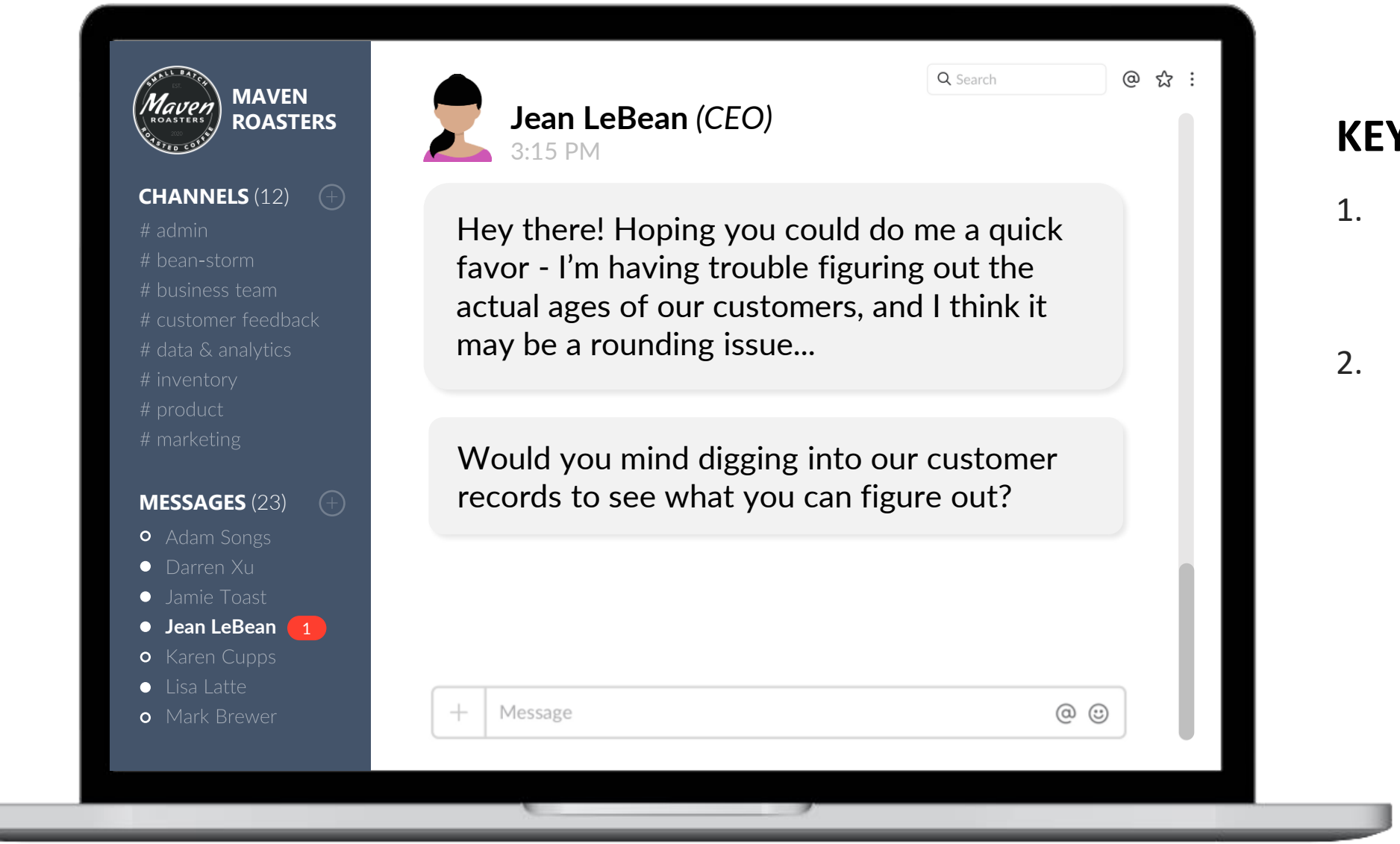
FLOOR *(9:34:14, “0:15”) =* ***9:30:00 AM***

*Rounds the minute component **down** to the nearest multiple*

CEILING *(9:34:14, ”0:15”) =* ***9:45:00 AM***

*Rounds the minute component **up** to the nearest multiple*

ASSIGNMENT: ROUNDING FUNCTIONS



KEY OBJECTIVES:

1. Add a new column in your **Customers** table to calculate age
2. Use a rounding function to make sure that ages are defined properly and formatted as whole numbers

INFORMATION FUNCTIONS

- Aggregation Functions
- Rounding Functions
- **Information Functions**
- Conversion Functions
- Logical Functions

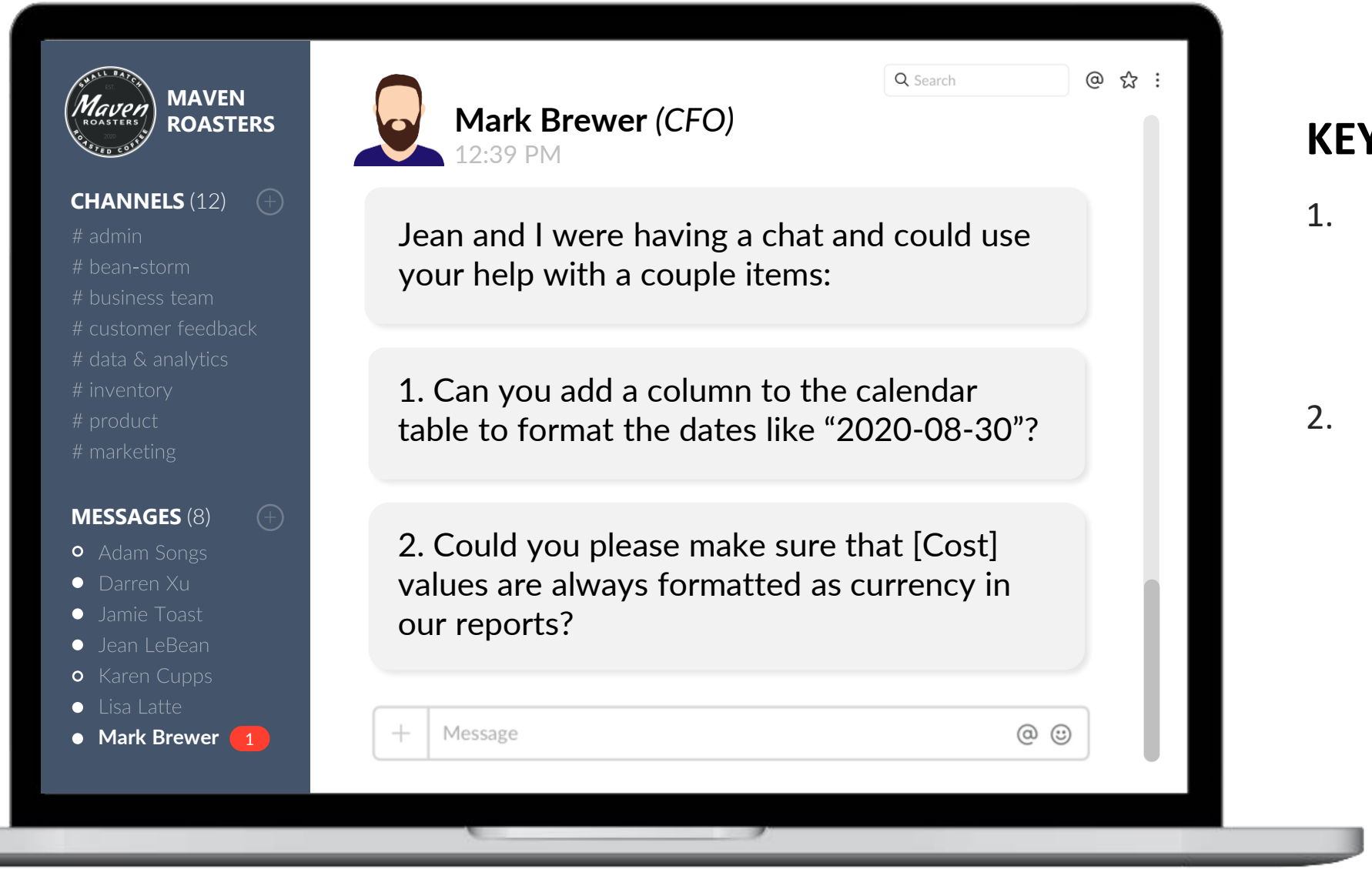
ISBLANK()	*Checks whether a value is blank, and returns TRUE or FALSE*	=**ISBLANK**(Value)
ISERROR()	*Checks whether a values is an error, and returns TRUE or FALSE*	=**ISERROR**(Value)
ISLOGICAL()	*Checks whether a value is a logical value (TRUE or FALSE), and returns TRUE or FALSE*	=**ISLOGICAL**(Value)
ISNUMBER()	*Checks whether a value is a number, and returns TRUE or FALSE*	=**ISNUMBER**(Value)
ISNONTEXT()	*Checks whether a value is not text (blank cells are not text), and returns TRUE or FALSE*	=**ISNONTEXT**(Value)
ISTEXT()	*Checks whether a value is text, and returns TRUE or FALSE*	=**ISTEXT**(Value)

CONVERSION FUNCTIONS

- Aggregation Functions
- Rounding Functions
- Information Functions
- **Conversion Functions**
- Logical Functions

CURRENCY()	*Evaluates the argument and returns the result as a currency data type*	=**CURRENCY**(Value)
FORMAT()	*Converts a value to text in the specified number format*	=**FORMAT**(Value, Format)
DATE()	*Returns the specified date in datetime format*	=**DATE**(Year, Month, Day)
TIME()	*Converts hours, minutes, and seconds given as numbers to a time in datetime format*	=**TIME**(Hours, Minute, Second)
DATEVALUE()	*Converts a date in the form of text to a date in datetime format*	=**DATEVALUE**(DateText)
VALUE()	*Converts a text string that represents a number to a number*	=**VALUE**(Text)

ASSIGNMENT: CONVERSION FUNCTIONS



KEY OBJECTIVES:

1. Create a new calculated column to convert **Transaction_Date** to yyyy-mm-dd format
2. Use a conversion function to modify the **[Cost]** measure and ensure that values are always displayed as currency *(without simply changing the format)*

BASIC LOGICAL FUNCTIONS

- Aggregation Functions
- Rounding Functions
- Information Functions
- Conversion Functions
- **Logical Functions**

IF() — *Checks if a given condition is met, and returns one value if the condition is TRUE, and another if the condition is FALSE*

=IF(LogicalTest, ResultIfTrue, *[ResultIfFalse]*)

AND() — *Checks whether both arguments are TRUE, and returns TRUE if both arguments are TRUE, otherwise returns FALSE*

=AND(Logical1, Logical2)

OR() — *Checks whether one of the arguments is TRUE to return TRUE, and returns FALSE if both arguments are FALSE*

=OR(Logical1, Logical2)

Note: *Use the* ***&&*** *and* ***||*** *operators if you want to include more than two conditions!*

SWITCH

Aggregation Functions

Rounding Functions

Information Functions

Conversion Functions

Logical Functions

SWITCH()

Evaluates an expression against a list of values and returns one of multiple possible expressions

=SWITCH(Expression, Value1, Result1, …, *[Else]*)

Any DAX expression that returns a single scalar value, evaluated multiples times

Examples:

- *Calendar[Month_ID]*
- *Product Lookup[product_group]*

*List of **values** produced by the expression, each paired with a result to return for rows/cases that match*

Examples:

=SWITCH(Calendar[Month_ID],

1, "January",

2. "February", etc.

Value returned if the expression doesn't match any value argument

PRO TIP

SWITCH (TRUE) is a common Pattern to replace nested IF statements

COALESCE

- Aggregation Functions
- Rounding Functions
- Information Functions
- Conversion Functions
- Logical Functions

COALESCE()

Returns the first argument that does not evaluate to BLANK. If all arguments evaluate to BLANK, BLANK is returned.

=COALESCE(Expression1, Expression2, *[…]*)

Any value or expression that returns a scalar value

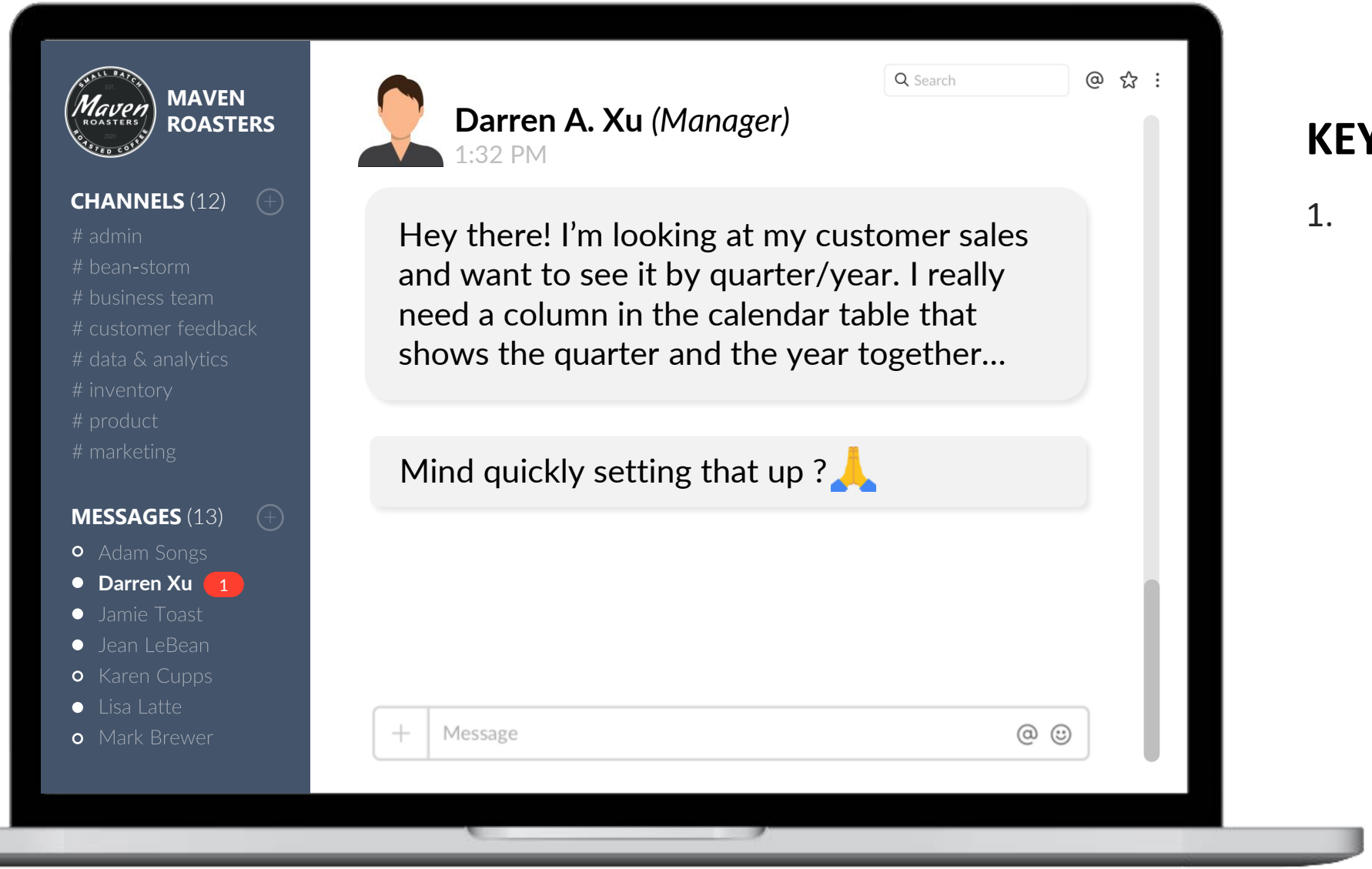
Examples:

- COALESCE(
  SUM('Sales by Store'[quantity_sold]),
  0)
- COALESCE(
  [Yesterday Customer Revenue],
  "-")

PRO TIP:

*COALESCE replaces the **IF + ISBLANK** pattern, makes your code more readable, and can be optimized by the engines*

ASSIGNMENT: LOGICAL FUNCTIONS



KEY OBJECTIVES:

1. Create a calculated column using **variables** and **SWITCH(TRUE())** to create a single column containing both quarter & year *(i.e. Q1-2020, Q2-2020, etc.)*

ADVANCED CALCULATE

CALCULATE is a powerful DAX function which is commonly used to modify filter context; in this section, we'll explore advanced topics like **expanded tables**, **context transition, evaluation order** and **modifiers**

TOPICS WE'LL COVER:

- **Expanded Tables**
- **Context Transition**
- **Evaluation Order**
- **Modifiers**

COMMON USE CASES:

- Simplifying measures by adding new filter context that would normally use multiple nested functions
- Modifying filter context to override preexisting filters using modifiers like ALL, KEEPFILTERS or REMOVEFILTERS
- Creating calculated inputs for measures like % of total, % of category, etc.

EXPANDED TABLES

- Expanded Tables
- Context Transition
- Evaluation Order
- Modifiers

An **expanded table** consists of the base table *(which is visible to the user)*, along with columns from any related table connected via a 1-to-1 or many-to-1 relationship

***Product Lookup** Table*

Product ID	Name	Category
1	Brazilian - Organic	Coffee beans
2	Old Time Diner	Coffee beans
3	Espresso Roast	Coffee beans
4	Primo Roast	Coffee beans
5	Columbian Medium	Coffee beans

***Store Lookup** Table*

Store ID	Address	Manager
3	32-20 Broadway	6
4	604 Union Street	11
5	100 Church Street	16
6	122 E Broadway	21
7	224 E 57th Street	26

1 → * (Product Lookup to Transactions)

1 → * (Store Lookup to Transactions)

Trans ID	Product ID	Store ID	Quantity	Name	Category	Address	Manager
1	4	4	32	Primo Roast	Coffee beans	604 Union Street	11
2	2	3	17	Old Time Diner	Coffee beans	32-20 Broadway	6
3	3	7	113	Espresso Roast	Coffee beans	224 E 57th Street	26
4	3	3	14	Espresso Roast	Coffee beans	32-20 Broadway	6
5	1	4	55	Brazilian - Organic	Coffee beans	604 Union Street	11

***Transactions** Table*

*Related fields from the **Product** and **Store** tables are accessible to DAX as part of the "expanded" **Transactions** table*

As you begin to write more complex DAX & CALCULATE statements, understanding expanded tables is critical!

EXPANDED TABLES

- Expanded Tables
- Context Transition
- Evaluation Order
- Modifiers

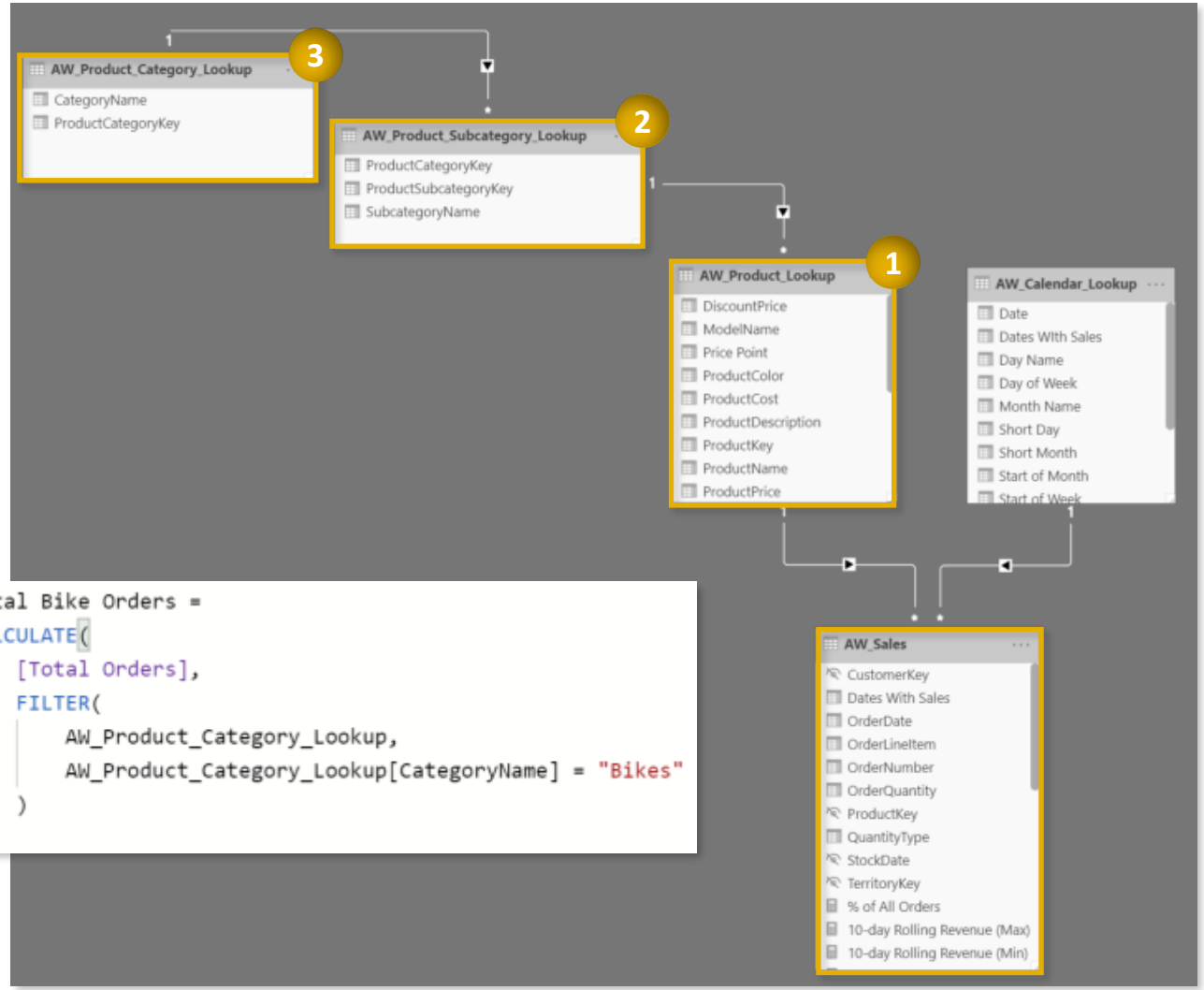
**Maven Roasters Data Model*

Expanded tables in Models *always* go from the **MANY** side to the **ONE** side of the relationship

- Expanded tables contain all the columns of the original table **plus** all the columns on the one-side
- Understanding expanded tables is useful because it provides a clear understanding of how filter context propagates in DAX measures
- Once a filter is applied to a column, **all expanded tables containing that column are also filtered**

EXPANDED TABLES (EXAMPLE)

- Expanded Tables
- Context Transition
- Evaluation Order
- Modifiers




```
Total Bike Orders =
CALCULATE(
    [Total Orders],
    FILTER(
        AW_Product_Category_Lookup,
        AW_Product_Category_Lookup[CategoryName] = "Bikes"
    )
)
```

**Adventure Works data model*

*In this case, the expanded version of the **AW_Sales** table contains all fields from the **product**, **subcategory**, and **category** lookup tables*

CONTEXT TRANSITION (CALCULATED COLUMNS)

Expanded Tables

Context Transition

Evaluation Order

Modifiers

Context Transition is the process of turning row context into filter context

- By default, calculated columns understand **row** context but not **filter** context
- To create filter context at the row-level, you can use **CALCULATE**

product_id	quantity_sold
70	9
70	4
70	1
70	2
70	5
70	6
70	8
70	7
70	3
70	11
70	10
70	12
70	15
70	13
70	14
70	16
70	17

Sum of Quantity
153
153
153
153
153
153
153
153
153
153
153
153
153
153
153
153
153

*If we add a column to calculate the sum of quantity_sold, we get repeating totals since there is **no filter context,** and we can't determine a unique value per row*

*To solve this, we can generate **filter context** at the **row-level**, using CALCULATE with SUMX*

```
1 Sum of Quantity =
2 CALCULATE(
3   SUM(
4      'Table'[quantity_sold])
5 )
```

CALCULATE Sum of Quantity
9
4
1
2
5
6
8
7
3
11
10
12
15
13
14
16
17

CONTEXT TRANSITION (MEASURES)

- Expanded Tables
- **Context Transition**
- Evaluation Order
- Modifiers

Consider the following formula, defined as both a **measure** and a **calculated column**:

```
1 Sum of Quantity Sold =
2 SUM('Table'[quantity_sold])
```

product_id	quantity_sold	Sum of Quantity	CALCULATE Sum of Quantity
69	3	959	3
69	1	959	1
69	2	959	2
69	5	959	5
69	6	959	6
69	8	959	8
69	7	959	7
72	20	959	20
72	18	959	18
72	17	959	17
72	21	959	21
72	19	959	19

*As a **calculated column**, we see the same total repeated on each row (no filter context)*

product_id	Sum of Quantity Sold	CALCUALTE Sum of Quantity Sold
69	154	154
70	153	153
71	326	326
72	326	326
Total	**959**	**959**

*As a **measure**, DAX automatically evaluates a CALCULATE & SUMX measure (shown below), to create the filter context needed to produce the correct values*

```
1 Sum of Quantity Sold =
2 CALCULATE(
3   SUMX(
4       'Table',
5       'Table'[quantity_sold])
6 )
```

EVALUATION ORDER (EXAMPLE)

- Expanded Tables
- Context Transition
- **Evaluation Order**
- Modifiers

***[Customer Sales]** is evaluated LAST, using the modified filter context*

***store_id = 3** is evaluated FIRST*

```
1  Store 3 Sales of Whole Bean/Teas (CALCULATE) =
2  CALCULATE(
3      3 [Customer Sales],
4      1 'Store Lookup'[store_id] = 3,
5      2 'Product Lookup'[product_group] = "Whole Bean/Teas"
6  )
```

***Product_group = "Whole Bean/Teas"** is evaluated SECOND*

***[Customer Sales]** is evaluated LAST, using the modified filter context*

***store_id** & **product_group** filters are evaluated next, and overwrite the ALL modifier*

```
1  Store 3 Sales of Whole Bean/Teas (Modifier) =
2  CALCULATE(
3      4 [Customer Sales],
4          2 'Store Lookup'[store_id] = 3,
5          3 'Product Lookup'[product_group] = "Whole Bean/Teas",
6          1 ALL(
7                'Store Lookup'
8            )
9  )
```

*ALL **Modifier** is evaluated FIRST*

Transaction_Date	Store 3 Sales of Whole Bean/Teas (CALCULATE)	Store 3 Sales of Whole Bean/Teas (Modifier)
1/8/2017	$89.80	$89.80
1/9/2017	$154.41	$154.41
1/10/2017	$61.65	$61.65
1/11/2017	$9.50	$9.50
1/12/2017	$84.20	$84.20
1/14/2017	$105.20	$105.20
1/15/2017	$257.03	$257.03
Total	**$761.79**	**$761.79**

*Both measures evaluate the same, since the ALL modifier evaluates first and is overwritten by the **store_id** and **product_group** filters*

HEY THIS IS IMPORTANT!

DAX engines combine multiple filter arguments into a *single* filter context

COMMON CALCULATE MODIFIERS

Expanded Tables

Context Transition

Evaluation Order

Modifiers

Modifiers are used to alter the way CALCULATE creates filter context, and are added as *filter* arguments within a CALCULATE function

- Modifiers are typically used to change filter context, access inactive table relationships, or change the way filters propagate *(i.e. one-way to bidirectional)*

Modify **Filters**

Common Examples:

- ALL
- ALLSELECTED
- ALLNOBLANKROW
- ALLEXCEPT
- KEEPFILTERS
- REMOVEFILTERS

Use **Relationships**

Common Examples:

- USERELATIONSHIP

Change **Filter Propagation**

Common Examples:

- CROSSFILTER

CALCULATE REVIEW (BOOLEAN & TABLE FILTERS)

Expanded Tables

Context Transition

Evaluation Order

Modifiers

=**CALCULATE**(Expression, *[Filter1], [Filter2],…*)

CALCULATE filter expressions accept both boolean & table functions *(individually or at the same time!),* but all filter arguments are automatically converted into a table

```
1  Store 3 Sales =
2  CALCULATE(
3      [Customer Sales],
4          'Store Lookup'[store_id] = 3
5  )
```

DAX interprets this as a table!

- Any time you use write a function that contains a logical statement (IN, >,<, =, etc.) you're **creating a table** *(internally processed with FILTER & ALL)*

```
1  All Store Sales =
2  CALCULATE(
3      [Customer Sales],
4      ALL(
5          'Store Lookup'
6      )
7  )
```

And this too, obviously 

REMOVEFILTERS

Expanded Tables

Context Transition

Evaluation Order

Modifiers

REMOVEFILTERS()

Clears filters from the specified tables or columns

=**REMOVEFILTERS** (TableNameorColumnName, *[ColumnName], […]*)

Table or column that you want to clear the filters from

Examples:

- *CALCULATE(*
  [Customer Sales],
  REMOVEFILTERS*(*
  'Store Lookup'))

*(Optional) Repeatable column names that allow you to specifically remove filters from **individual columns** within the base table*

Examples:

- *CALCULATE(*
  [Customer Sales],
  REMOVEFILTERS*(*
  'Store Lookup'[store_id]))

PRO TIP:

REMOVEFILTERS is an alias for ALL, but can only be used as a CALCULATE modifier (not as a table function)

KEEPFILTERS

Expanded Tables

Context Transition

Evaluation Order

Modifiers

KEEPFILTERS()

Does not remove an existing column or table filter for an individual CALCULATE expression

=**KEEPFILTERS**(Expression)

CALCULATE *or* ***CALCULATETABLE*** *function expression or filter*

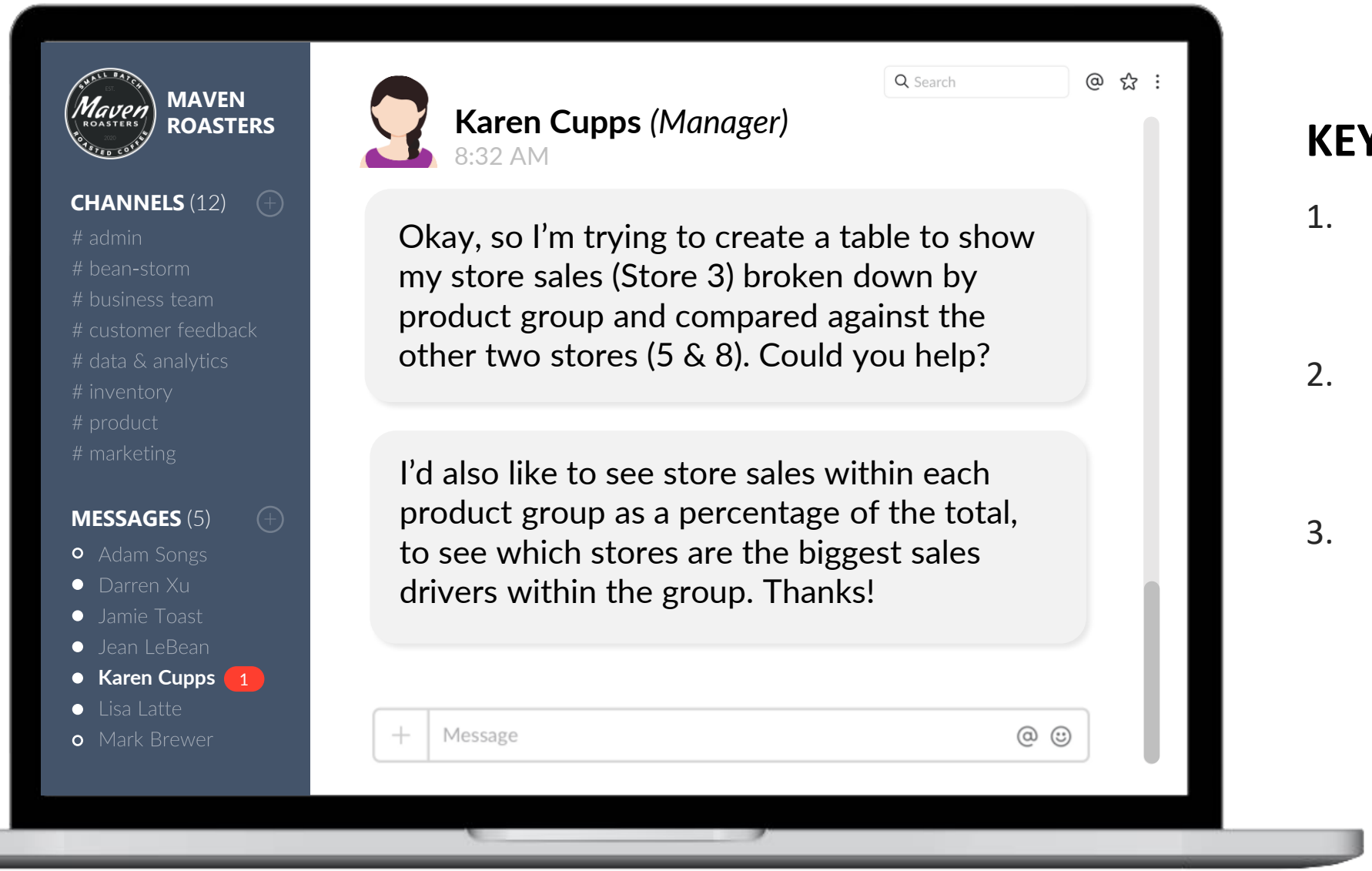
Examples:

- *CALCULATE(*
  [Measure],
  KEEPFILTERS*('Table'[ID]="Value"))*

✓ Allows you to control **which specific filters** are applied to a calculation

✓ KEEPFILTERS does not actually *modify* existing filters, but adds new filter context *(think **inner join**)*

ASSIGNMENT: ADVANCED CALCULATE



KEY OBJECTIVES:

1. Create a matrix containing **product group** & **store id** on rows
2. Use **KEEPFILTERS** to create measures that show sales for each of the 3 stores
3. Use **REMOVEFILTERS** with **variables** to create a single measure for *% of Store Sales*

PRO TIP: COMMON CALCULATE TOTAL PATTERNS

Cumulative Total

```
Cumulative Total =
CALCULATE(
    SUM(
        'Sales by Store'[quantity_sold]
    ),
    FILTER(
        ALL(
            'Calendar'[Transaction_Date]
        ),
        'Calendar'[Transaction_Date] <= MAX('Calendar'[Transaction_Date])
    )
)
```

Overall Total

```
Overall Total =
CALCULATE(
    SUMX(
        'Sales by Store',
        'Sales by Store'[unit_price] * 'Sales by Store'[quantity_sold]
    ),
    ALL(
        'Sales by Store'
    )
)
```

Percent of Total

```
Percent of Total Sales =
VAR CurrentSales =
SUMX(
    'Sales by Store',
    'Sales by Store'[unit_price] * 'Sales by Store'[quantity_sold]
)
VAR ALLStoreSales =
CALCULATE(
    SUMX(
        'Sales by Store',
        'Sales by Store'[unit_price] * 'Sales by Store'[quantity_sold]
    ),
    ALL(
        'Sales by Store'
    )
)
VAR Ratio =
DIVIDE(
    CurrentSales,
    ALLStoreSales
)
RETURN
Ratio
```

TABLE & FILTER FUNCTIONS

TABLE & FILTER FUNCTIONS

Table and **Filter** functions return **columns** or **tables** rather than scalar values, and can be used to either generate new data or serve as table inputs within DAX measures

TOPICS WE'LL COVER:

- **Calculated Tables**
- **Filtering Tables**
- **Manipulating Tables**
- **Generating Data**

COMMON USE CASES:

- *Returning columns or tables which are subsets of existing data sources*
- *Adding new columns of data to an existing table*
- *Filtering out totals from measure results*
- *Reducing the number of rows returned from a table*
- *Generating new data from scratch*

COMMON TABLE & FILTER FUNCTIONS

Filter Data

Functions used to return filtered tables or filter results of measures

Common Examples:

- ALL
- FILTER
- DISTINCT
- VALUES
- ALLEXCEPT
- ALLSELECTED

Add Data

Functions used to specify or add columns based on existing data in the model

Common Examples:

- SELECTCOLUMNS
- ADDCOLUMNS
- SUMMARIZE

Create Data

Functions used to generate new rows, columns & tables from scratch

Common Examples:

- ROW
- DATATABLE
- GENERATESERIES
- { } Table Constructor

REVIEW: CALCULATED TABLES

- Calculated Tables
- Filtering Tables
- Manipulating Tables
- Generating Data

DAX functions with **table** arguments can typically accept either **physical** tables *(i.e. 'Sales by Store')* or **calculated, virtual** tables *(with functions like FILTER, VALUES, etc.)*

=**SUMX**(Table, Expression)

```
1 Total Sales =
2 SUMX(
3     'Sales by Store',
4     'Sales by Store'[quantity_sold])
```

```
1 Total Sales =
2 SUMX(
      FILTER(
3         'Sales by Store',
4         'Sales by Store'[quantity_sold]) > 3
5     ),
6     'Sales by Store'[quantity_sold]) )
```

REVIEW: FILTER & ALL

- Calculated Tables
- Filtering Tables
- Manipulating Tables
- Generating Data

FILTER()

Returns a filtered table, based on one or more filter expressions

=**FILTER**(Expression, *[Filter1], [Filter2],…*)

- FILTER is both a **table function** and an **iterator**
- Often used to **reduce** the number of rows to scan

ALL()

Returns all the rows in a table, or all the values in a column, ignoring any filters

=**ALL**(Table *or* ColumnName, *[ColumnName1],…*)

- ALL is both a **table filter** and a **CALCULATE modifier**
- Removes initial **filter context**
- Does not accept **table expressions** *(only physical table references)*

REVIEW: FILTER (EXAMPLE)

- Calculated Tables
- Filtering Tables
- Manipulating Tables
- Generating Data

STEP 1 ***Measure is written***

```
1 Sales Where Order Quantity More Than 3 =
2 CALCULATE(
3     [Customer Sales],
4     FILTER(
5         'Sales by Store',
6         'Sales by Store'[Quantity_Sold] >3
7     )
8 )
```

=

$29,643

Sales Where Order Quantity More Than 3

STEP 3 ***DAX evaluates [Customer Sales] against the filtered table (quantity sold > 3)***

STEP 2 ***FILTER creates a virtual, calculated table***

```
1 Sales greater than 3 =
2 FILTER(
3     'Sales by Store',
4     'Sales by Store'[Quantity_Sold] > 3
5 )
```

Transaction_ID	Transaction_Date	Transaction_Time	Store_ID	Employee_ID	Customer_Number	In-Store_Purchase	Order	Line_Item_ID	Product_ID	Quantity_Sold	Line_Item_Amount	Unit_Price	Promo_Item
405	1/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64				
405	3/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64				
405	3/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64				
284	4/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63				
284	4/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63				
405	5/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64				
284	6/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63				
284	6/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63				
284	6/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63				
284	6/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63				
405	6/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64				
405	6/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64				
405	6/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64				
284	7/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63	4	$3.2	$0.8	N
405	7/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64	4	$3.2	$0.8	N
284	8/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63	4	$3.2	$0.8	N
405	8/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64	4	$3.2	$0.8	N
405	8/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64	4	$3.2	$0.8	N
405	8/8/2017 12:00:00 AM	12/30/1899 7:32:15 PM	5	29	0	N	1	2	64	4	$3.2	$0.8	N
284	9/8/2017 12:00:00 AM	12/30/1899 6:36:57 PM	5	30	0	N	1	2	63	4	$3.2	$0.8	N

Sort ascending
Sort descending
Clear sort
Clear filter
Clear all filters
Number filters
Search
(Select all)
4
6
8
OK
Cancel

DISTINCT

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

DISTINCT()

*Returns a **single column** table of unique values when a **column name** is given. If a **table** is supplied, DISTINCT returns all unique combinations of values.*

=DISTINCT(ColumnName or TableExpression)

*The **column** you want to extract unique values from*

Examples:

- *'Sales by Store'[CustomerID]*
- *'Sales by Store'[UnitPrice]*

*The **table** you want to extract unique combinations of values from*

Examples:

- *'Sales by Store'*
- *'Food Inventory'*

PRO TIP:

*Trying to build a relational model from a single, blended table? Use **DISTINCT** to create new lookup/dimension tables by extracting unique values from fields in your data table!*

VALUES

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

VALUES()

*Returns a **single column** table of unique values when a **column name** is given. If a table name is supplied, VALUES returns the entire table (including duplicates) plus a blank row*

=**VALUES**(TableName or ColumnName)

*The **table** you want to pull all columns and all rows from (not unique)*

Examples:

- *'Sales by Store'*
- *'Food Inventory'*

*The **column** you want to extract unique values from*

Examples:

- *'Sales by Store'[CustomerID]*
- *'Sales by Store'[UnitPrice]*

VALUES VS. DISTINCT: THE BLANK ROW

Product Lookup

Product ID	Product
12	Coffee
13	Tea
14	Pastry
blank	blank

1

*

Sales by Store

Product ID	Quantity
12	2
13	6
14	1
14	3
15	5
16	6

*In this example we have a Product Lookup table containing three product IDs (**12, 13, 14**)*

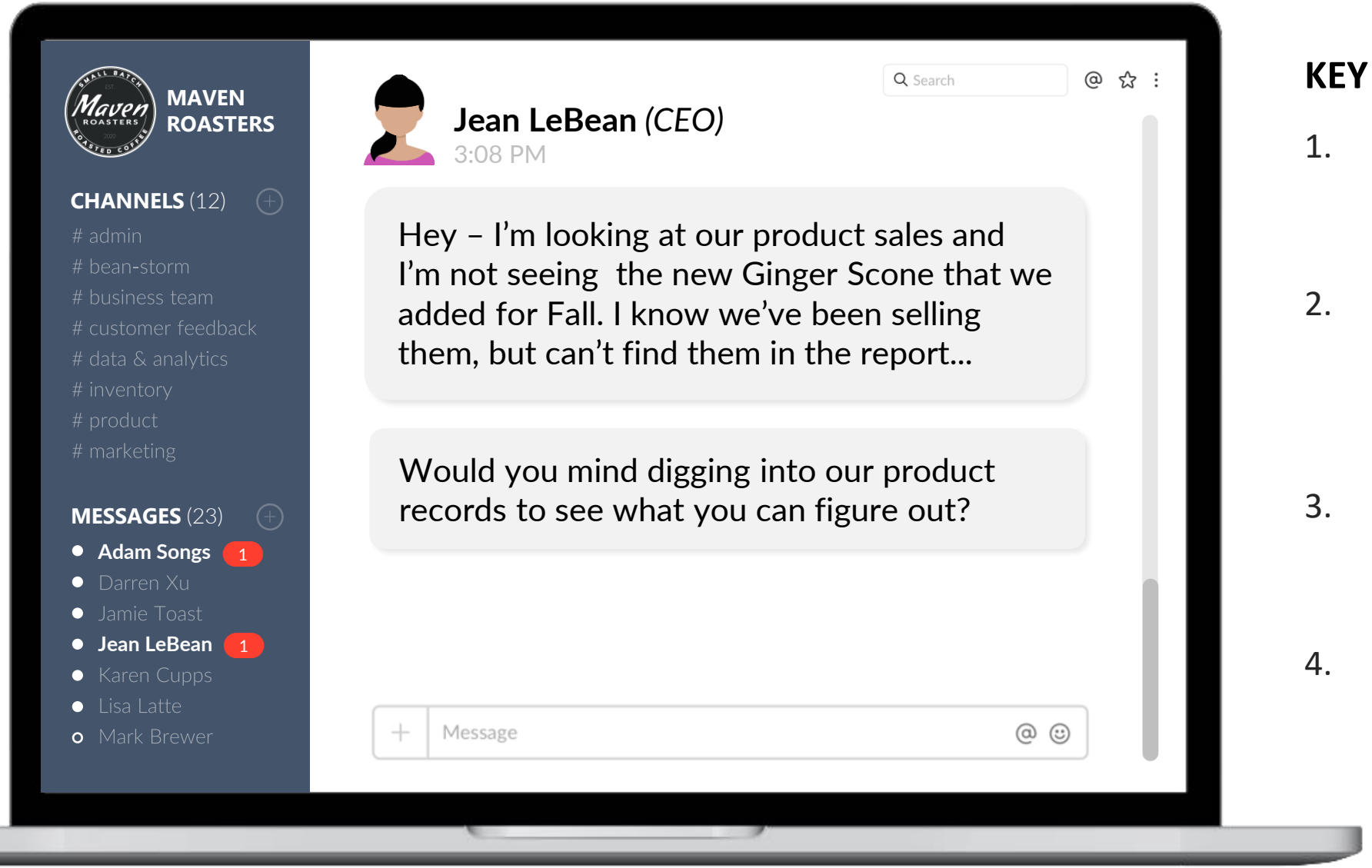
*But our fact table ('Sales by Store') contains additional product IDs NOT included in the Product Lookup **(15, 16)***

- Instead of throwing an error when a value is missing from a lookup table, the DAX engine adds a **blank row** *(which will appear in visuals when missing values are present)*
- Different table functions handle the presence of this blank row differently; for example, **VALUES** will always show the blank row but **DISTINCT** will not

PRO TIP:

*If you think you might have missing values in your lookup tables (or aren't sure), use **VALUES** to check*

ASSIGNMENT: VALUES & DISTINCT BLANK ROW



KEY OBJECTIVES:

1. Check if the Ginger Scone product exists in the data model
2. Use counting functions plus DISTINCT & VALUES to create a view that shows the blank row
3. Add a new visual to show the missing product id when the blank product is selected
4. Update the data model to include the new product

Hint: *Product Lookup (Updated)*

SELECTEDVALUE

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

SELECTEDVALUE()

*Returns a value when there's **only one value** in a specified column, otherwise returns an (optional) alternate result*

=**SELECTEDVALUE**(ColumnName, [AlternateResult])

The column you want to return a single value from

Examples:

- *'Customer Lookup'[customer_first_name]*
- *'Product Lookup'[product]*

***(Optional)** The value returned when there is either no value or more than one value in the specified column (if omitted, blank is returned)*

Examples:

- *"-"*
- *"NA"*

product_group	Customer Sales	Quantity Sold (SELECTEDVALUE)	Retail Price (SELECTEDVALUE)
Add-ons	**$51,060.00**		
Carmel syrup	$12,366.40	15,458.00	$0.80
Chocolate syrup	$12,840.80	16,895.79	$0.76
Hazelnut syrup	$11,641.60	14,925.13	$0.78
Sugar Free Vanilla syrup	$14,211.20	17,330.73	$0.82
Beverages	**$3,282,118.55**		
Food	**$501,291.32**		
Merchandise	**$83,784.00**		
Whole Bean/Teas	**$334,451.01**		
Total	**$4,252,704.88**		

*Because the expression doesn't evaluate to a **single** (scalar) result, nothing is returned. In this case, only when a single retail price can be determined is a result returned.*

PRO TIP: SELECTEDVALUE "SYNTAX SUGAR"

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

SELECTEDVALUE is another example of "syntax sugar" in DAX

- The DAX engine internally processes **SELECTEDVALUE** as a combination of **IF**, **HASONEVALUE**, and **VALUES**:

How it's written:

```
1 SELECTEDVALUE =
2  SELECTEDVALUE(
3       Customer[customer_first-name],
4       "-"
5  )
```

How it's interpreted:

```
1 SELECTEDVALUE =
2  IF(
3     HASONEVALUE(
4       Customer[customer_first-name]),
5        VALUES(
6        Customer[customer_first-name]),
7        "-"
8  )
```

ALLEXCEPT

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

ALLEXCEPT()

*Removes all report context filters in the table **except** the filters applied to the specified columns in the query*

=ALLEXCEPT(TableName, [ColumnName],[ColumnName], […])

*Name of an **existing** table (the use of table expressions is not allowed here)*

Examples:

- *'Sales by Store'*
- *'Returns'*
- *FILTER(*
  'Sales by Store',
  'Sales by Store'[store_id] IN {2,5,7,9,10})

*Additional column references within the **same referenced table** or a table that is on the **one-side of the relationship** (adding additional columns is optional)*

Examples:

- *'Sales by Store'[product_group]*
- *'Sales by Store'[product_category]*
- *'Calendar'[Transaction_Date]*

PRO TIP:

***ALLEXCEPT** is typically used as a CALCULATE modifier and not a stand-alone table function*

ALLEXCEPT (EXAMPLE)

Calculated Tables

Filtering Tables

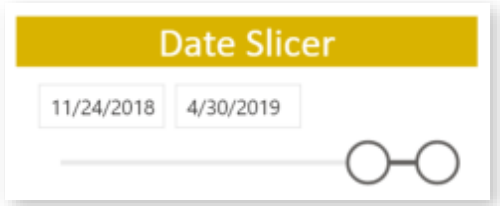
Manipulating Tables

Generating Data

```
1 ALLEXCEPT Demo =
2 CALCULATE(
3     [Customer Sales],
4     ALLEXCEPT(
5         'Sales by Store',
6         'Calendar'[Transaction_Date], 'Store Lookup'[sales_outlet_id]
7     )
8 )
```

*Here we're using **ALLEXCEPT** as a CALCULATE modifier to remove all initial filter context from the '**Sales by Store**' table, except for filters on the following columns:*

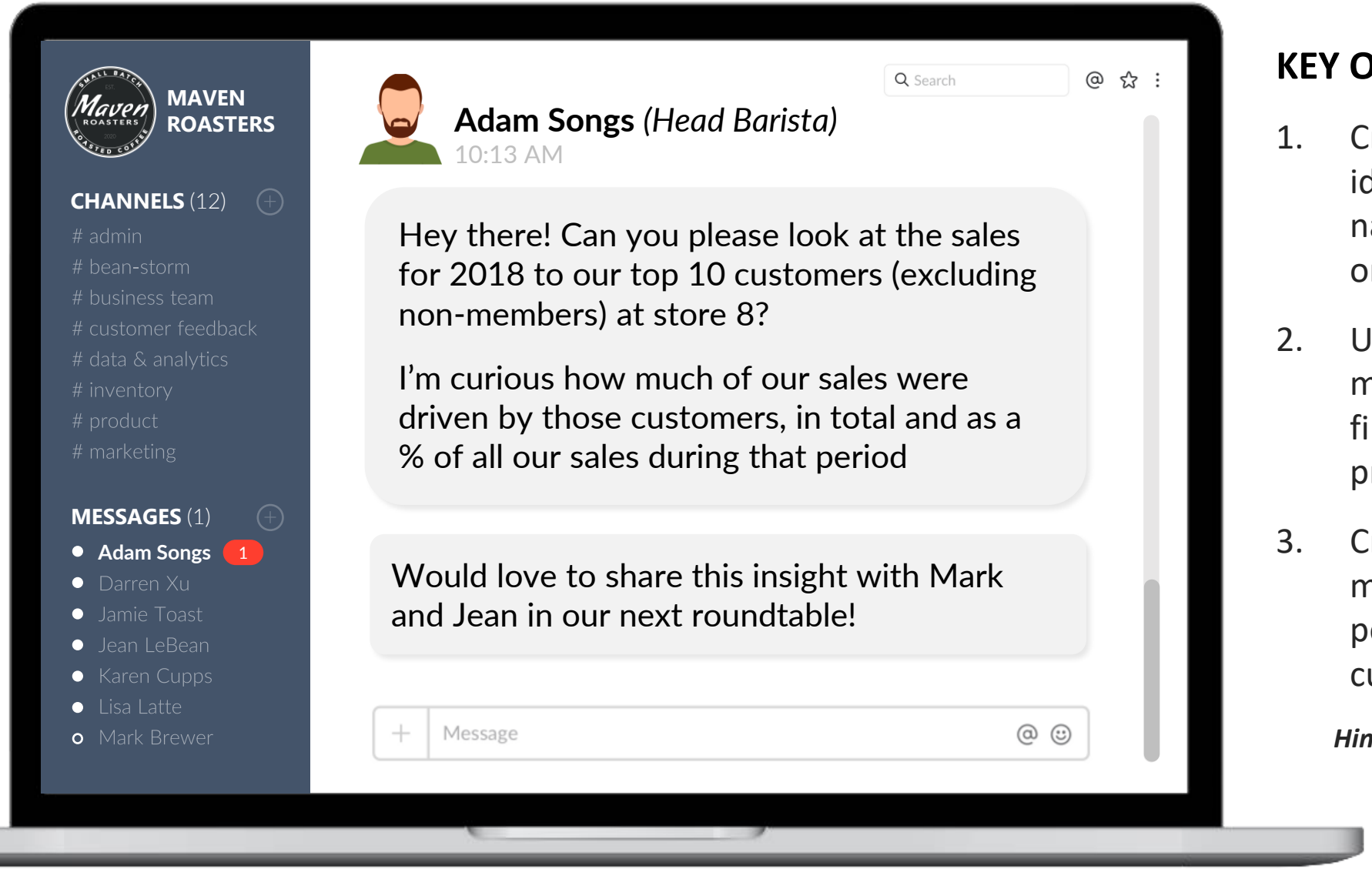
- *'Calendar'[Transaction_Date]*
- *'Store Lookup'[sales_outlet_id]*



store_id	Customer Sales	Percent of All Selected Sales	ALLEXCEPT Demo
3	$295,632.11	33.30%	$295,632.11
Add-ons	$2,142.40	0.24%	$295,632.11
Beverages	$234,583.35	26.42%	$295,632.11
Food	$33,571.50	3.78%	$295,632.11
Bakery	$33,571.50	3.78%	$295,632.11
Merchandise	$6,727.00	0.76%	$295,632.11
Whole Bean/Teas	$18,607.86	2.10%	$295,632.11
5	$293,310.74	33.04%	$293,310.74
8	$298,812.44	33.66%	$298,812.44
Total	$887,755.29	100.00%	$887,755.29

*The measure above returns Customer Sales by **store ID** and **Date**, but ignores filter context created by other fields (i.e. Product Category)*

ASSIGNMENT: ALLEXCEPT



KEY OBJECTIVES:

1. Create a matrix with the store id, product group and customer name on rows, filtered based on Adam's request
2. Use **ALLEXCEPT** to create a measure that only accepts filters for the date, store ID product group and customer
3. Create a **% of store-level sales** measure to calculate the percent of sales by the top 10 customers at the store level

Hint: *Try REMOVEFILTERS + KEEPFILTERS*

ALLSELECTED

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

ALLSELECTED()

Returns all rows in a table or values in a column, ignoring filters specified in the query but keeping any other existing filter context

=ALLSELECTED(TableNameOrColumnName, [ColumnName],[ColumnName], […])

*Name of a table or column that you want to remove filters from (**NOTE**: This input is required unless being used as a CALCULATE modifier)*

Examples:

- *ALLSELECTED*
- *'Sales by Store'*
- *'Sales by Store'[product_group]*

*Additional column references within the **same** referenced table (adding additional columns is optional)*

Examples:

- *'Sales by Store'[product_group]*
- *'Sales by Store'[product_category]*

HEY THIS IS IMPORTANT!

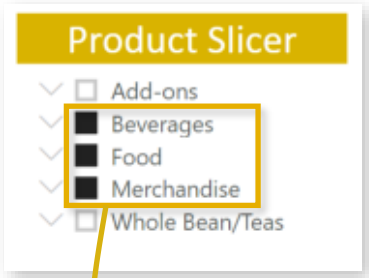
ALLSELECTED respects existing filter context **except** *row* and *column* filters within a visual. This function can be used to obtain visual totals or subtotals in queries.

ALLSELECTED (EXAMPLE)

- Calculated Tables
- **Filtering Tables**
- Manipulating Tables
- Generating Data

```
1  Selected Sales (ALLSELECTED) =
2  CALCULATE(
3      [Customer Sales],
4      ALLSELECTED(
5      )
6  )
```

*This measure calculates **sales** (quantity * price), in a modified filter context based on external filters*

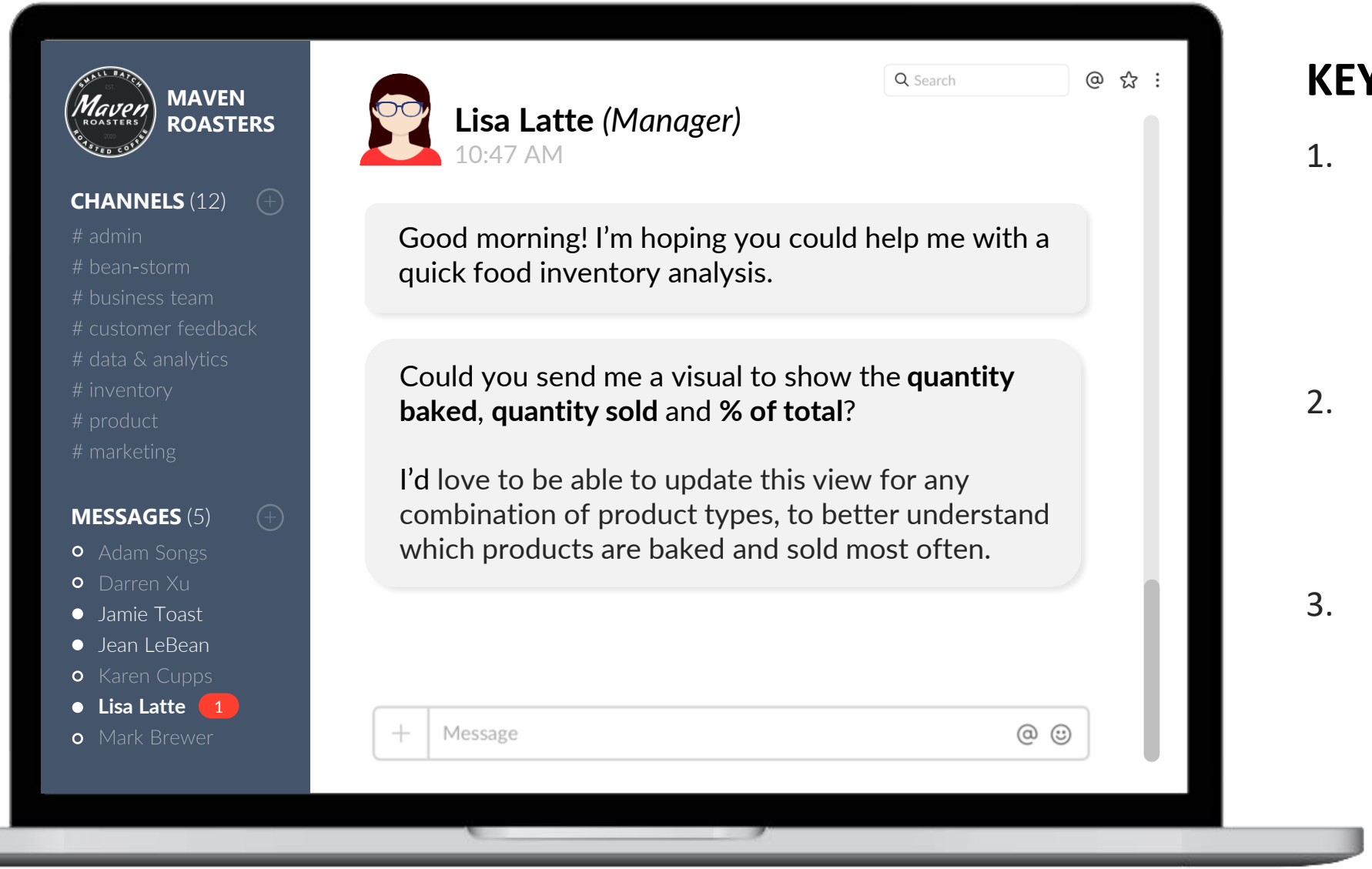


product_group	Customer Sales	All Sales	Percent of All Sales	Selected Sales (ALLSELECTED)	Percent of All Selected Sales
Beverages	$3,282,118.55	$4,252,704.88	77.18%	$3,867,193.87	84.87%
Food	$501,291.32	$4,252,704.88	11.79%	$3,867,193.87	12.96%
Merchandise	$83,784.00	$4,252,704.88	1.97%	$3,867,193.87	2.17%
Total	**$3,867,193.87**	**$4,252,704.88**	**90.93%**	**$3,867,193.87**	**100.00%**

*When we select specific product groups in a report slicer (**Beverages**, **Food** & **Merchandise**) ALLSELECTED respects the filter context from the selection (slicer) but not from the rows and columns of the visual.*

This function is commonly used to obtain visual totals or subtotals in queries

ASSIGNMENT: ALLSELECTED



KEY OBJECTIVES:

1. Using the Food Inventory table, create two measures for **Total Baked** *(quantity start of day)* and **Total Sold** *(quantity sold)*
2. Create measures for **% of Total Baked** and **% of Total Sold**, totaling 100% based on any selected product types
3. Build a visual to compare the **% of All Baked** to **% of Total Baked**. Which food item is baked the most?

SELECTCOLUMNS

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

SELECTCOLUMNS()

Returns a table with selected columns from the table plus any new columns specified by the DAX expressions

=SELECTCOLUMNS(Table, Name, Expression, […])

Any DAX expression that returns a table

***Examples**:*

- *'Sales by Store'*
- *'Food Inventory'*
- *FILTER(*
  'Employees',
  'Employees'[ID] IN {2,5,7,9,10})

Name of the new column to be added, must be wrapped with double quotes. Repeatable

***Examples**:*

- *"Employee Name & ID"*
- *"Employee Full Name"*

Any expression that returns a scalar value (i.e. column reference, integer or string). Repeatable

***Examples**:*

- *'Employees'[ID]*
- *[First Name] & " " & [Last Name]*

PRO TIP:

SELECTCOLUMNS is an iterator function that's useful when you need to reduce the number of columns in a table for calculation

ADDCOLUMNS

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

ADDCOLUMNS()

Returns a table with selected columns from the table plus any new columns specified by the DAX expressions

=ADDCOLUMNS(Table, Name, Expression, […])

Any DAX expression that returns a table

***Examples**:*

- *'Sales by Store'*
- *'Food Inventory'*
- *FILTER(*
  'Employees',
  'Employees'[ID] IN {2,5,7,9,10})

Name of the new column to be added, must be wrapped with double quotes. Repeatable

***Examples**:*

- *"Employee Name & ID"*
- *"Employee Full Name"*

Any expression that returns a scalar value (i.e. column reference, integer or string). Repeatable

***Examples**:*

- *'Employees'[ID]*
- *[First Name] & " " & [Last Name]*

HEY THIS IS IMPORTANT!

ADDCOLUMNS & SELECTCOLUMNS are nearly identical and behave similarly with an exception, SELECTCOLUMNS starts with a blank table whereas ADDCOLUMNS starts with the entire original table and tacks on columns

SUMMARIZE

- Calculated Tables
- Filtering Tables
- **Manipulating Tables**
- Generating Data

SUMMARIZE()

Creates a summary of the input table grouped by the specified columns

=SUMMARIZE(Table, GroupBy_ColumnName, [Name], [Expression])

***Depreciated** and not recommended*

Any DAX expression that returns a table of data

***Examples**:*

- *'Sales by Store'*
- *FILTER(*
  'Sales by Store',
  'Sales by Store'[product_id] = 16)

Name of an existing column to be used to create summary groups based on the values found in the column. Cannot be an expression

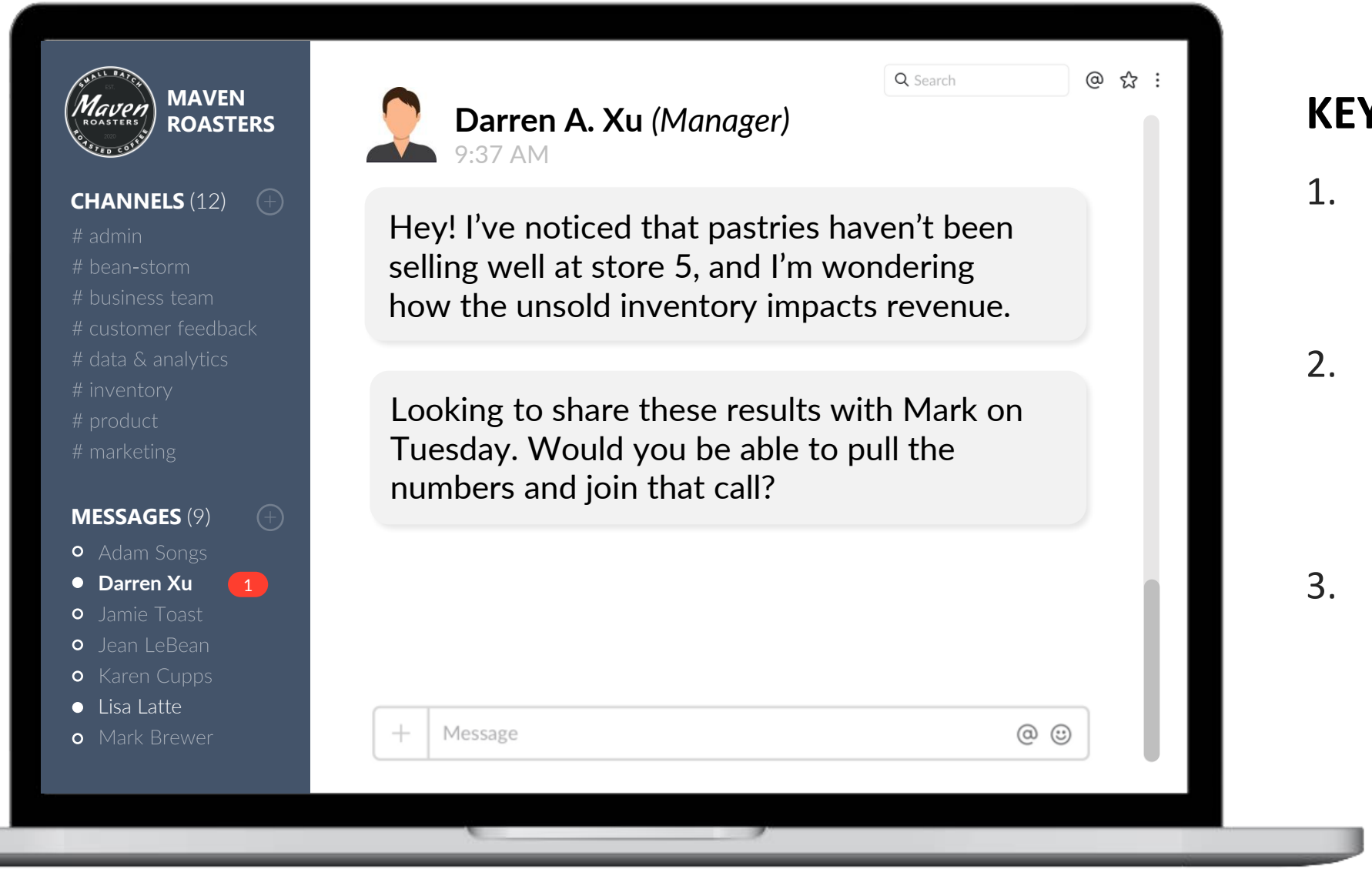
***Examples**:*

- *'Sales by Store'[store_id]*
- *'Customer Lookup'[customer_id]*

HEY THIS IS IMPORTANT!

SUMMARIZE isn't an aggregation function. Instead it returns all unique combinations of values based on the columns selected

ASSIGNMENT: SUMMARIZE



KEY OBJECTIVES:

1. Use **SUMMARIZE** to create a table that only shows the days with unsold pastries
2. Include columns to calculate amount sold, amount unsold, and total lost revenue
3. Build a matrix to visualize the results for Darren and Mark

GENERATING ROWS & TABLES OF DATA

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

In this section, we'll review **four methods** of manufacturing data from scratch:

ROW()	*Returns a single row table with new columns specified by the DAX expression(s)*	=**ROW**(Name, Expression, [...]])
DATATABLE()	*Returns a table containing new, static data*	=**DATATABLE**(Name, Type, [...]], Data)
GENERATESERIES()	*Returns a single column table containing sequential values, based on a given increment*	=**GENERATESERIES**(StartValue, EndValue, *[IncrementValue]*)
Table Constructor { }	*Uses curly brackets to return a table containing one or more columns and records*	= { scalarExpr1, scalarExpr2, … } OR = { (scalarExpr1, scalarExpr2, …), (scalarExpr1, scalarExpr2, …), }

ROW

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

ROW()

Returns a table with a single row containing values that result from the expressions given to each column

=**ROW**(Name, Expression, [Name, Expression, […]])

Name of the new column, enclosed in double quotes. Repeatable.

***Examples**:*

- *"Profit"*
- *"Cost"*

Any DAX expression that returns a single scalar value

***Examples**:*

- *[Customer Sales]*
- *SUM(*
  'Food Inventory'[quantity_start_of_day]

```
1  ROW Example =
2  ROW(
3      "Customer Sales", [Customer Sales],
4      "Items Ordered", [Quantity Sold]
5  )
```

Customer Sales	Items Ordered
4252704.88	1305637

DATATABLE

- Calculated Tables
- Filtering Tables
- Manipulating Tables
- Generating Data

DATATABLE() *Returns a table with a fixed set of values*

=**DATATABLE**(Name, Type, [Name, Type], […]], Data)

Column name (in quotes)

***Examples**:*

- *"Test Number"*
- *"Value"*

Column data type

***Examples**:*

- *STRING*
- *DOUBLE*
- *INT*
- *CURRENCY*

*Data points to add to table, called and separated with curly brackets (**{ }**)*

***Examples**:*

- *{"First", 1},*
- *{"Second", 2},*
- *{"Third", 3}*

HEY THIS IS IMPORTANT!

DATATABLE can only accept **fixed data inputs** *(no table or column references, expressions or calculations)*

GENERATESERIES

- Calculated Tables
- Filtering Tables
- Manipulating Tables
- Generating Data

GENERATESERIES() — *Returns a one column table populated with sequential values*

=**GENERATESERIES**(StartValue, EndValue, *[IncrementValue]*)

Start & End value for the series (can be positive or negative)

***Examples**:*

- *0, 1, 10.25, 50.234, etc.*
- *-1, -10.75, -45.5, etc.*

*Value used to increment between Start & End value (must be **non-zero** and **positive**)*

***Examples**:*

- *.25, 1, 1.7745, 5, etc.*

PRO TIP:

***GENERATESERIES** is a great way to build a custom range of data to be used as a parameter input*

TABLE CONSTRUCTOR

Calculated Tables

Filtering Tables

Manipulating Tables

Generating Data

The **table constructor { }** can be used to build tables containing one or more columns or rows, based on fixed values or DAX expressions which return scalar values

- ***NOTE:*** *Column headers can't be defined using table constructor syntax*

1 *Single **column** with **multiple rows**:*

```
1  Table Constructor Single Column Table =
2  {
3      [Customer Sales],
4      [Quantity Sold]
5  }
```

Value
4252704.88
1305637

2 *Single **row** with **multiple columns**:*

```
1  Table Constructor Single Row Table =
2  {
3      (
4      [Customer Sales],
5      [Quantity Sold]
6      )
7  }
```

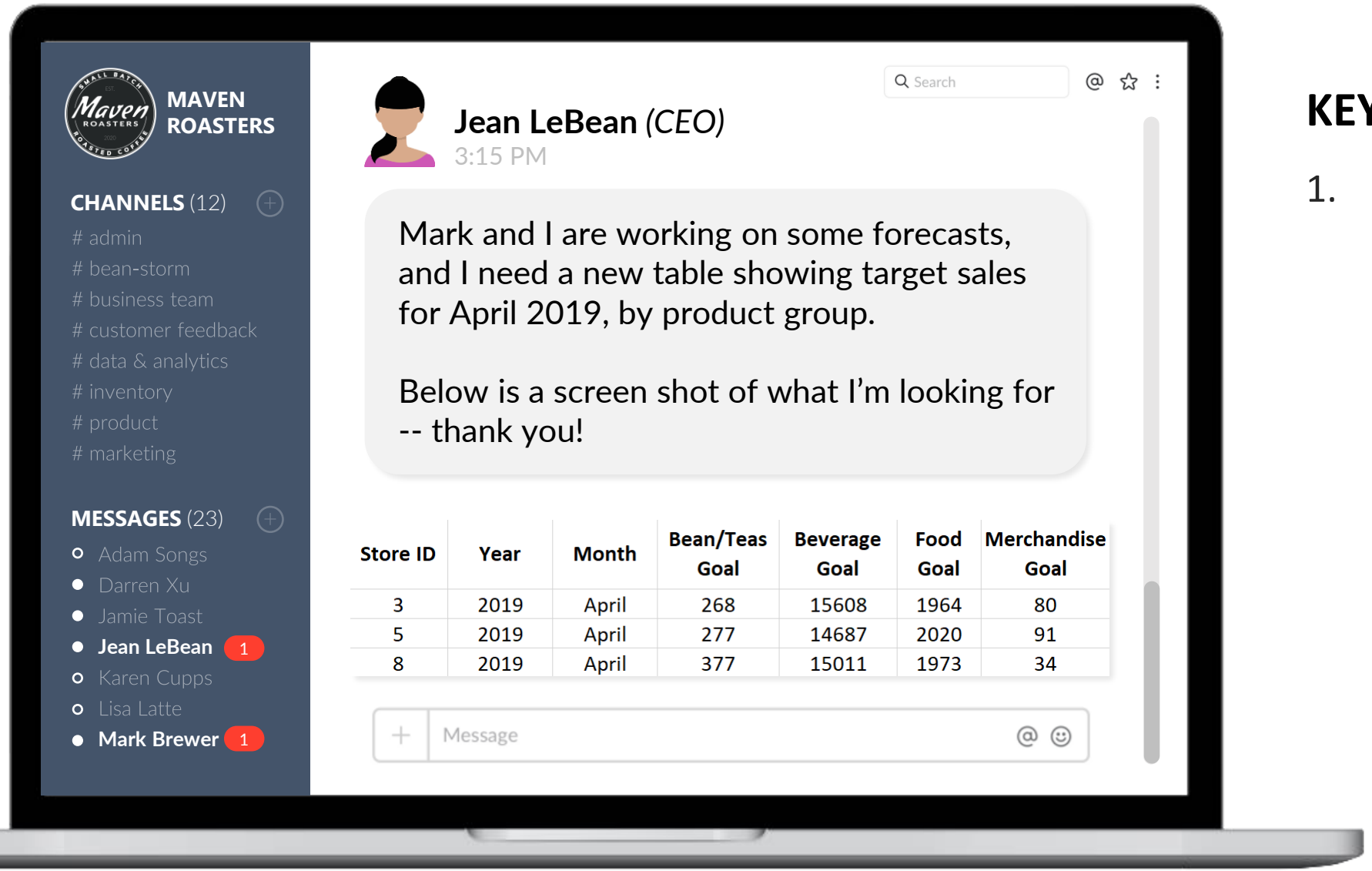
Value1	Value2
4252704.88	1305637

3 ***Multi-row, multi-column** table:*

```
1  Table Constructor Example =
2  {
3      ("This Is the First Column", -- Column 1 Row 1
4      "This is the Second Column"  -- Column 2 Row 1
5      ),
6      ([Customer Sales],           -- Column 1 Row 2
7      [Quantity Sold]              -- Column 2 Row 2
8      ),
9      ([Sales to Females],         -- Column 1 Row 3
10     [Count of Store Sales]       -- Column 2 Row 3
11     )
12 }
```

Value1	Value2
This Is the First Column	This is the Second Column
4252704.88	1305637
224	30

ASSIGNMENT: GENERATING DATA



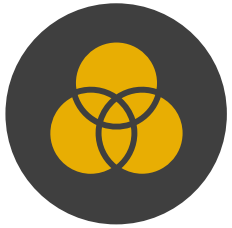
Store ID	Year	Month	Bean/Teas Goal	Beverage Goal	Food Goal	Merchandise Goal
3	2019	April	268	15608	1964	80
5	2019	April	277	14687	2020	91
8	2019	April	377	15011	1973	34

KEY OBJECTIVES:

1. Use **DATATABLE** to recreate the table that Jean needs for her forecasts

CALCULATED TABLE JOINS

CALCULATED TABLE JOINS



Calculated table joins are used to combine two or more tables of data; common examples include **CROSSJOIN**, **UNION**, **EXCEPT** and **INTERSECT**

TOPICS WE'LL COVER:

- **CROSSJOIN**
- **UNION**
- **EXCEPT**
- **INTERSECT**

COMMON USE CASES:

- *Blending or combining data across multiple tables*
- *Creating advanced calculations like new vs. returning users or repeat purchase behavior*
- *Querying tables to troubleshoot errors or better understand relationships in a data model*

CROSSJOIN

CROSSJOIN

UNION

EXCEPT

INTERSECT

CROSSJOIN()

Product of two sets, forming a new set containing all ordered pairs

Returns a table that contains the cartesian product of the specified tables

=CROSSJOIN(Table, Table, *[…]]*)

List of table expressions to include in the crossjoin

***Examples**:*

- *'Product Lookup'*
- *VALUES(*
  'Sales by Store'[store_id))

Product Category	Product Group
Coffee	Beverages
Tea	Beverages
Bakery	Food
Chocolate	Food

Store ID
3
5
8

CROSSJOIN

Product Category	Product Group	Store ID
Coffee	Beverages	3
Tea	Beverages	3
Bakery	Food	3
Chocolate	Food	3
Coffee	Beverages	5
Tea	Beverages	5
Bakery	Food	5
Chocolate	Food	5
Coffee	Beverages	8
Tea	Beverages	8
Bakery	Food	8
Chocolate	Food	8

*Resulting table contains **12 rows** (4*3) and **3 columns** (2+1)*

IMPORTANT NOTES:

- ✓ Column names must all be **different** in all table arguments
- ✓ The # of rows returned equals the **product of rows** in all tables
- ✓ The # of columns returned equals the **sum of columns** in all tables

UNION

CROSSJOIN

UNION

EXCEPT

INTERSECT

UNION() — *Combines or "stacks" rows from two or more tables sharing the same column structure*

=UNION(Table, Table, *[…]*)

*Accepts any DAX expression for two (or more) tables with **identical column structure***

***Examples**:*

- *'Sales Target 2019', 'Sales Target 2020'*
- *'Sales Target 2019', DATATABLE( )*

Date	ABC	XYZ	123
5/1/19	A	X	1
5/1/19	B	Y	2
5/1/19	C	Z	3

Date	ABC	XYZ	123
5/1/20	A	X	1
5/1/20	B	Y	2
5/1/20	C	Z	3

UNION

Date	ABC	XYZ	123
5/1/19	A	X	1
5/1/19	B	Y	2
5/1/19	C	Z	3
5/1/20	A	X	1
5/1/20	B	Y	2
5/1/20	C	Z	3

***UNION** stacks tables together, just like **append**!*

IMPORTANT NOTES:

- ✓ All tables must contain **the same number of columns**
- ✓ Columns are combined by **position** in their respective tables
- ✓ Column names are determined by the **first table expression**
- ✓ **Duplicate rows** are retained

EXCEPT

CROSSJOIN

UNION

EXCEPT

INTERSECT

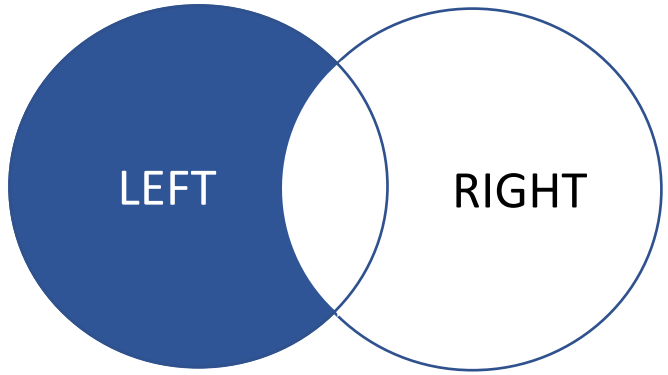
EXCEPT() *Returns all rows from the left table which do not appear in the right table*

=**EXCEPT** (LeftTable, RightTable)

The left and right tables that will be used for joining

***Example** (churned customers):*

=EXCEPT('Customers 2019','Customers 2020')



*Resulting table contains rows which ONLY appear in the **left table***

IMPORTANT NOTES:

- ✓ Both tables must contain **the same number of columns**
- ✓ Columns are compared based on **positioning** in their respective tables
- ✓ Column names are determined by the **left table**
- ✓ The resulting table does NOT retain **relationships** to other tables *(can't be used as an expanded table)*

INTERSECT

CROSSJOIN

UNION

EXCEPT

INTERSECT

INTERSECT() *Returns all the rows from the left table which also appear in the right table*

=INTERSECT(LeftTable, RightTable)

The left and right tables that will be used for joining

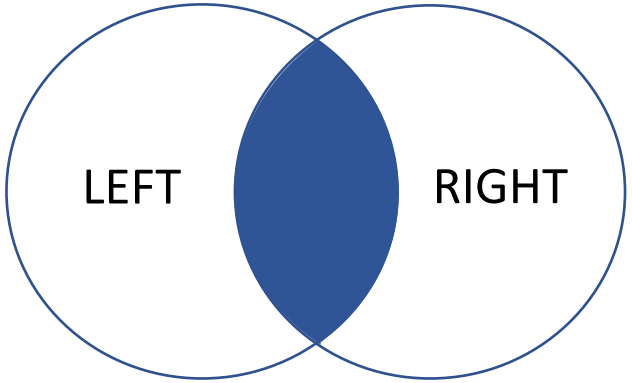
***Example** (previous month active customers):*

LeftTable:

```
VALUES('Sales'[Customer ID])
```

RightTable:

```
CALCULATETABLE(
    VALUES('Sales'[Customer ID]),
    DATEADD('Calendar'[Date],-1, MONTH))
```

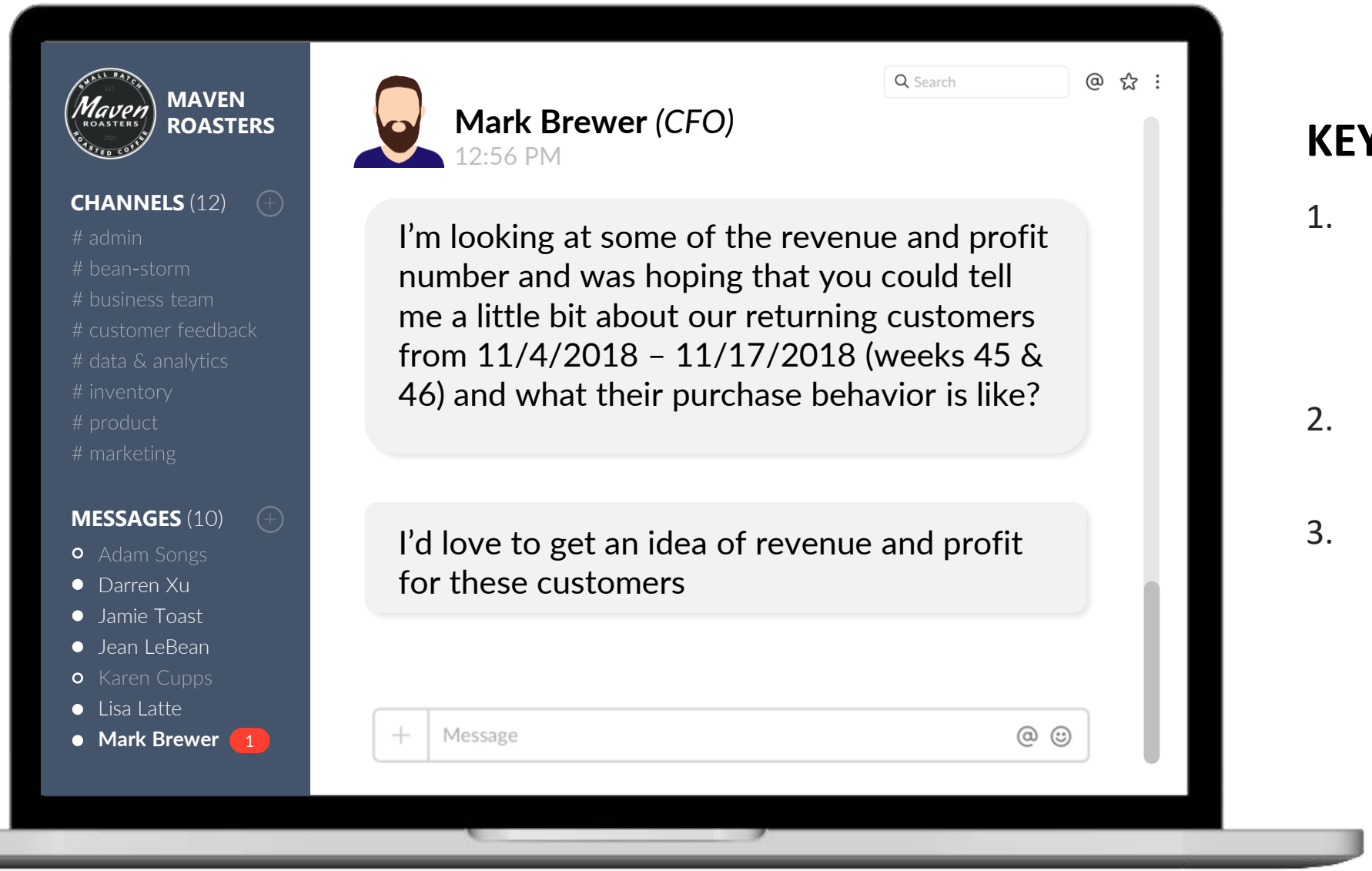


*Resulting table contains rows which appear in **BOTH** tables*

IMPORTANT NOTES:

- ✓ **Order matters!** *(T1, T2)* may have a different result set than *(T2, T1)*
- ✓ Columns are compared based on **positioning** in their respective tables
- ✓ **Duplicate** rows are retained
- ✓ Column names are determined by the **left table**
- ✓ The resulting table does NOT retain **relationships** to other tables *(can't be used as an expanded table)*

ASSIGNMENT: JOINS



KEY OBJECTIVES:

1. Use **CALCULATETABLE** & **INTERSECT** to look at returning customers between 11/4/2018 & 11/17/2018
2. Add additional columns for Revenue and Profit
3. Determine the total Revenue and Profit for repeat customers on week 46

RELATIONSHIP FUNCTIONS

RELATIONSHIP FUNCTIONS

Relationship functions allow you to access fields within DAX measures or calculated columns through either *physical* or *virtual* relationships between tables

TOPICS WE'LL COVER:

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

COMMON USE CASES:

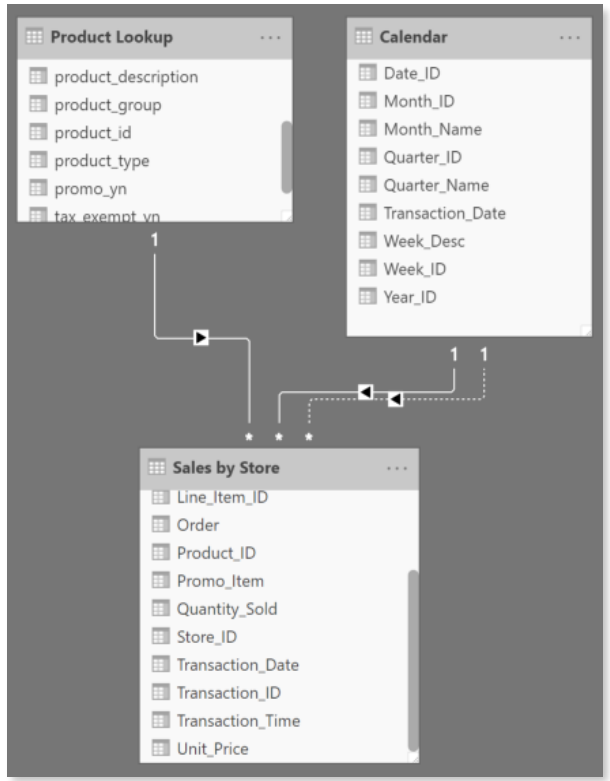
- *Defining calculated columns or measures using fields from related tables*
- *Handling relationships between calendars and multiple date fields (transaction date, stock date, due date, etc.)*
- *Traversing inactive relationships in the model, or modifying relationship filter behavior*
- *Defining calculated, virtual relationships when physical relationships aren't possible due to table granularity*

PHYSICAL VS. VIRTUAL RELATIONSHIPS

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

There are two key types of table relationships: **PHYSICAL** and **VIRTUAL**

- **Physical** relationships are manually created, and visible in your data model
- **Virtual** relationships are temporary, and defined using DAX expressions



These are **physical** relationships:

- *Visible links between tables (typically 1:* cardinality)*
- *Can be **active** or **inactive***
- *Can be accessed using DAX functions like **RELATED**, **RELATEDTABLE** or **USERELATIONSHIP***
- *Best way to connect tables (but not always possible)*

PHYSICAL VS. VIRTUAL RELATIONSHIPS

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

There are two key types of table relationships: **PHYSICAL** and **VIRTUAL**

- **Physical** relationships are manually created, and visible in your data model
- **Virtual** relationships are temporary, and defined using DAX expressions

```
1  Bean Goal (TREATAS) =
2  CALCULATE(
3      SUM(
4          'Target Sales UNION Example'[Bean/Teas Goal]
5      ),
6      TREATAS(
7          VALUES(
8              'Calendar'[Year_ID]
9          ),
10         'Target Sales UNION Example'[Year]
11     ),
12     TREATAS(
13         VALUES(
14             'Calendar'[Month_Name]
15         ),
16         'Target Sales UNION Example'[Month]
17     )
18 )
```

This is a **virtual** relationship:

- *Defined using DAX expressions*
- *Used when a physical relationship doesn't exist, or cannot be created directly*
- *Often used to connect tables with different levels of **granularity** (i.e. daily sales vs. monthly budgets/goals)*
- *Can be accessed using DAX functions like **TREATAS***

RELATED

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

RELATED()

Returns a value from a related table in the data model

=RELATED(ColumnName)

*Name of the column you want to retrieve values from (must reference a table on the **"one"** side of a many-to-one relationship)*

Examples:

- *'Product Lookup'[current_cost]*
- *'Customer Lookup'[home_store]*

PRO TIP

*The **RELATED** function doesn't really perform an operation, it just "opens the door" to access columns from an expanded table*

RELATEDTABLE

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

RELATEDTABLE()

Returns a related table, filtered so that it only includes the related rows

=**RELATEDTABLE**(Table)

*Physical table you want to retrieve rows from (must reference a table on the **"many"** side of a many-to-one relationship)*

***NOTE:** RELATEDTABLE is commonly used with aggregators like **COUNTROWS**, **SUMX**, **AVERAGEX**, etc.*

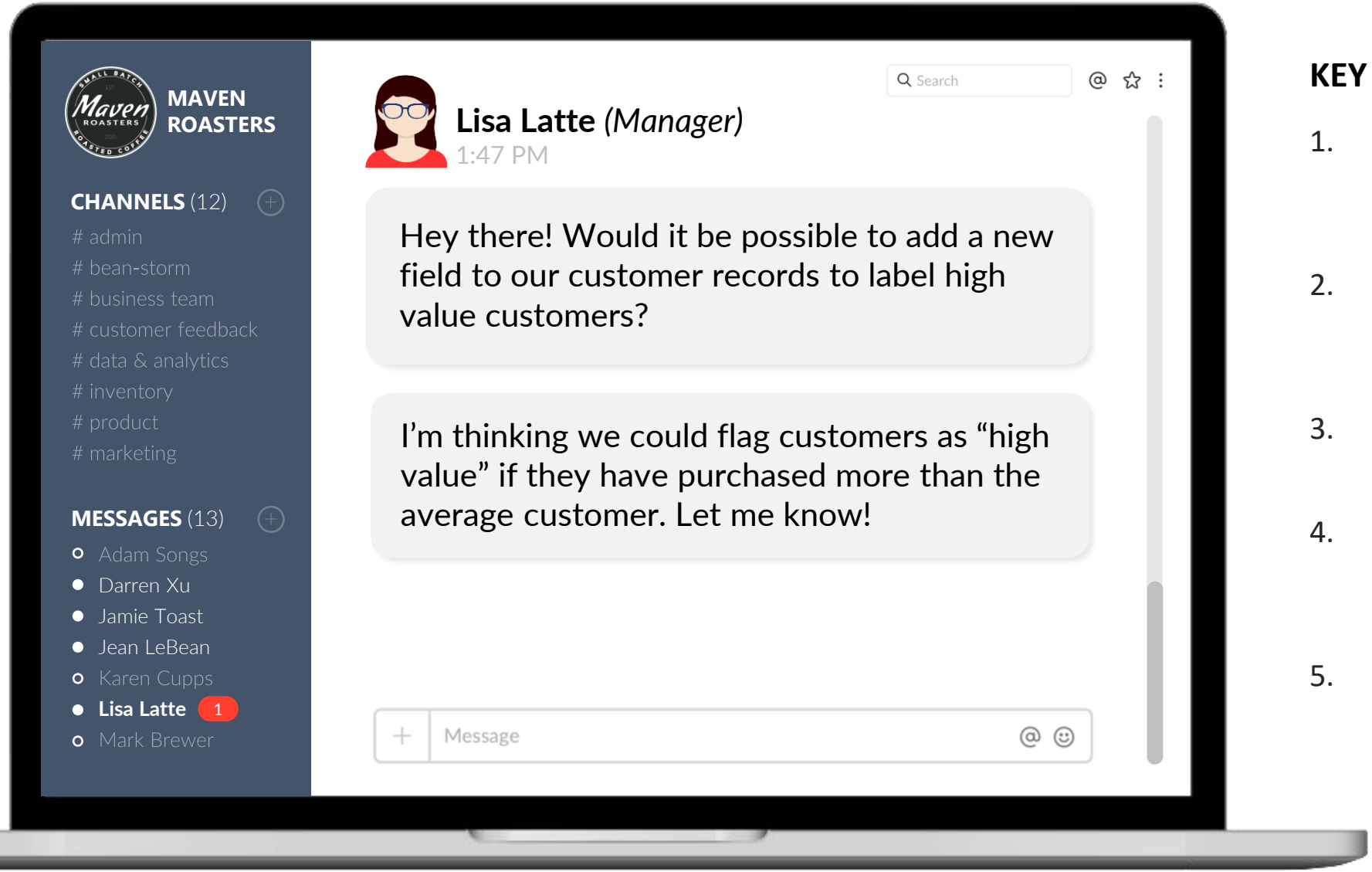
***Examples**:*

- *COUNTROWS(*
  RELATEDTABLE('Food Inventory'))
- *SUMX(*
  RELATEDTABLE('Food Inventory'),
  [Quantity Sold] * [Retail Price])

HEY THIS IS IMPORTANT!

RELATEDTABLE is a shortcut for CALCUATETABLE (with no logical expression) and performs a **context transition** from *row* context to *filter* context, in order to return only the rows which satisfy the filter condition(s)

ASSIGNMENT: RELATEDTABLE



KEY OBJECTIVES:

1. Add a calculated column to the Customer Lookup table and create a variable for "Total Sales"
2. Add another variable called "AllCustomers" to count total customers
3. Add another variable that defines average sales
4. Use RELATEDTABLE to create a variable that computes sales at the customer level
5. Return "High" if customer-level sales are above the average. Otherwise, return "Low"

USERELATIONSHIP

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

USERELATIONSHIP()

Specifies an existing relationship to be used in the evaluation of a DAX expression, defined by naming, as arguments, the two columns that serve as endpoints

=USERELATIONSHIP(ColumnName1, ColumnName2)

***Foreign** (or primary) key of the relationship*

Examples:

- *Food Inventory[Baked_Date]*
- *Calendar[Transaction Date]*

***Primary** (or foreign) key of the relationship*

Examples:

- *Calendar[Transaction Date]*
- *Food Inventory[Baked_Date]*

HEY THIS IS IMPORTANT!

USERELATIONSHIPS can only be used in functions which accept a filter parameter *(CALCULATE, TOTALYTD, etc.)*

PRO TIP:

*If you have **multiple date columns** connected to a single calendar table, USERELATIONSHIP is a great way to force measures to use **inactive relationships** without having to manually activate them in your model*

CROSSFILTER

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

CROSSFILTER()

Specifies cross filtering direction to be used for the duration of the DAX expression. The relationship is defined by naming the two columns that serve as endpoints

=CROSSFILTER(LeftColumnName, RightColumnName2, CrossFilterType)

*The two columns you want to use. **Left column** is typically the "**many**" side and **right column** is typically the "**one**" side*

Examples:

- *'Sales by Store'[customer_id]*
- *'Customer Lookup'[customer_id]*

Specifies the direction of the CROSSFILTER

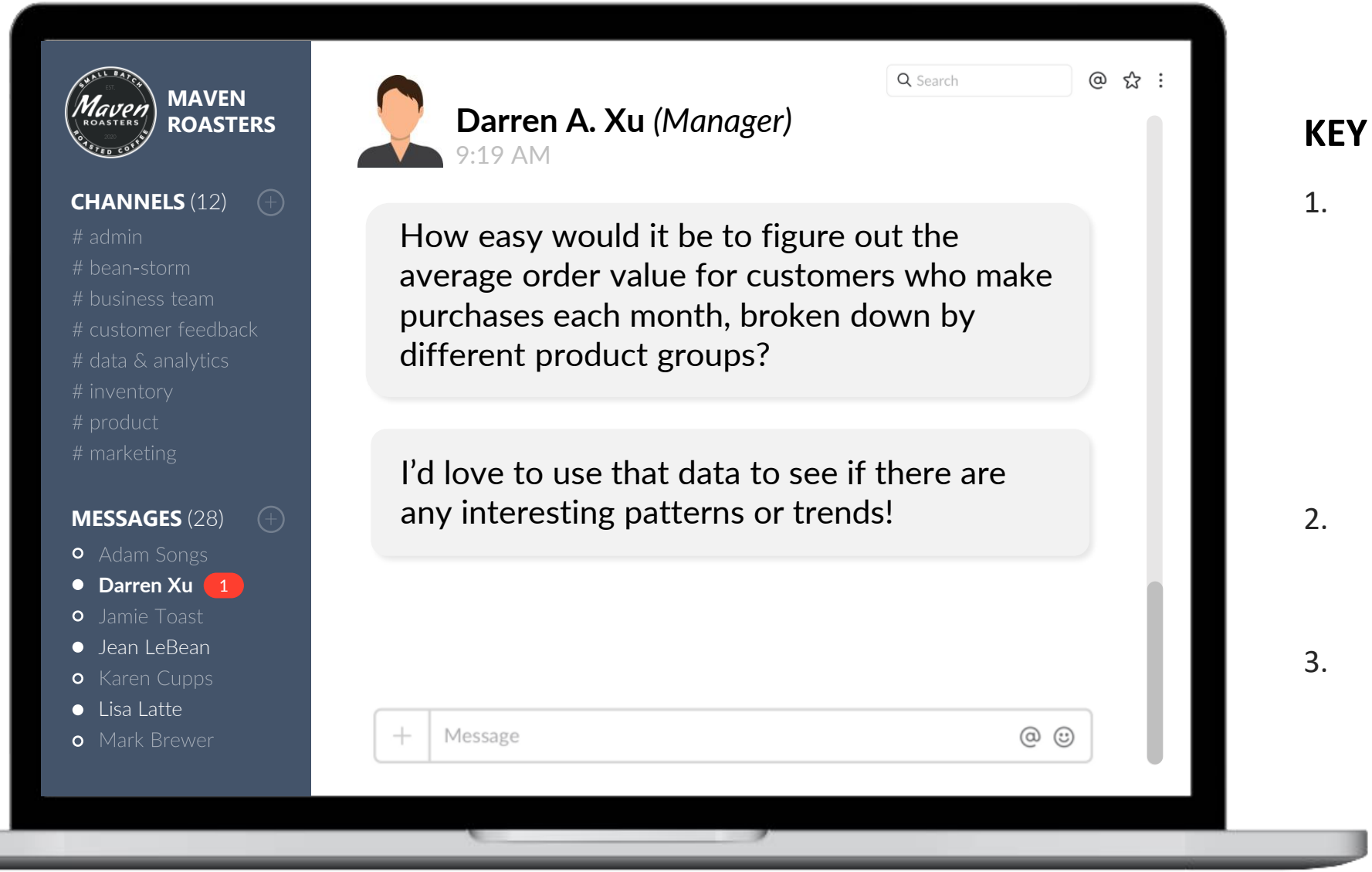
Examples:

- *OneWay, Both, None*

PRO TIP

*Instead of bi-directional relationships, use CROSSFILTER to enable two-way filtering **ONLY in specific cases***

ASSIGNMENT: CROSSFILTER



KEY OBJECTIVES:

1. Create a measure called "Customers who Purchased" that uses **COUNTROWS** & **CROSSFILTER** to calculate the number of customers who made a purchase in a given time period

   Hint: *Think about filter direction*

2. Create a measure that calculates the average order value for "Customers who Purchased"

3. Create a matrix that that shows the previous two measures broken down by Darren's request

TREATAS

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS

TREATAS()

Applies the result of a table expression to filter columns in an unrelated table (essentially creating a new, virtual relationship)

=**TREATAS**(TableExpression, ColumnName, *[ColumnName], […]*)

A table expression which generates the set of columns to be mapped. Table expression must be based on physical table in data model.

***Examples**:*

- ***TREATAS**(*
  VALUES('Calendar'[Year_ID])…
- ***TREATAS**(*
  SUMMARIZE('Calendar',
  'Calendar'[Year_ID], 'Calendar[Month]…

The list of output columns (cannot be an expression)

NOTE:** The number of columns specified must match the **number of columns** in the table expression and be in the **same order

***Examples**:*

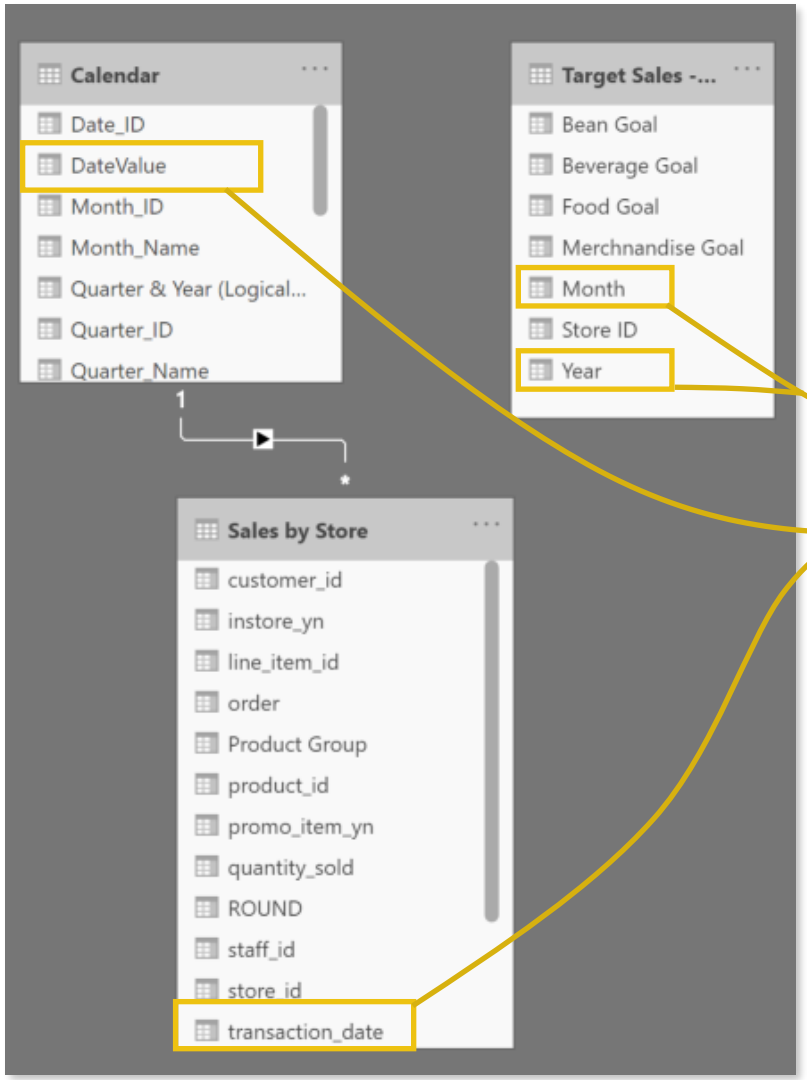
- *'Target Sales'[Year]*
- *'Target Sales'[Year], 'Target Sales'[Month]*

PRO TIP:

*Use **physical** relationships (or **USERELATIONSHIP** functions) whenever possible, and only rely on TREATAS if you are unable to create a direct relationship between tables*

TREATAS (EXAMPLE)

- Physical vs. Virtual Relationships
- RELATED
- RELATEDTABLE
- USERELATIONSHIP
- CROSSFILTER
- TREATAS



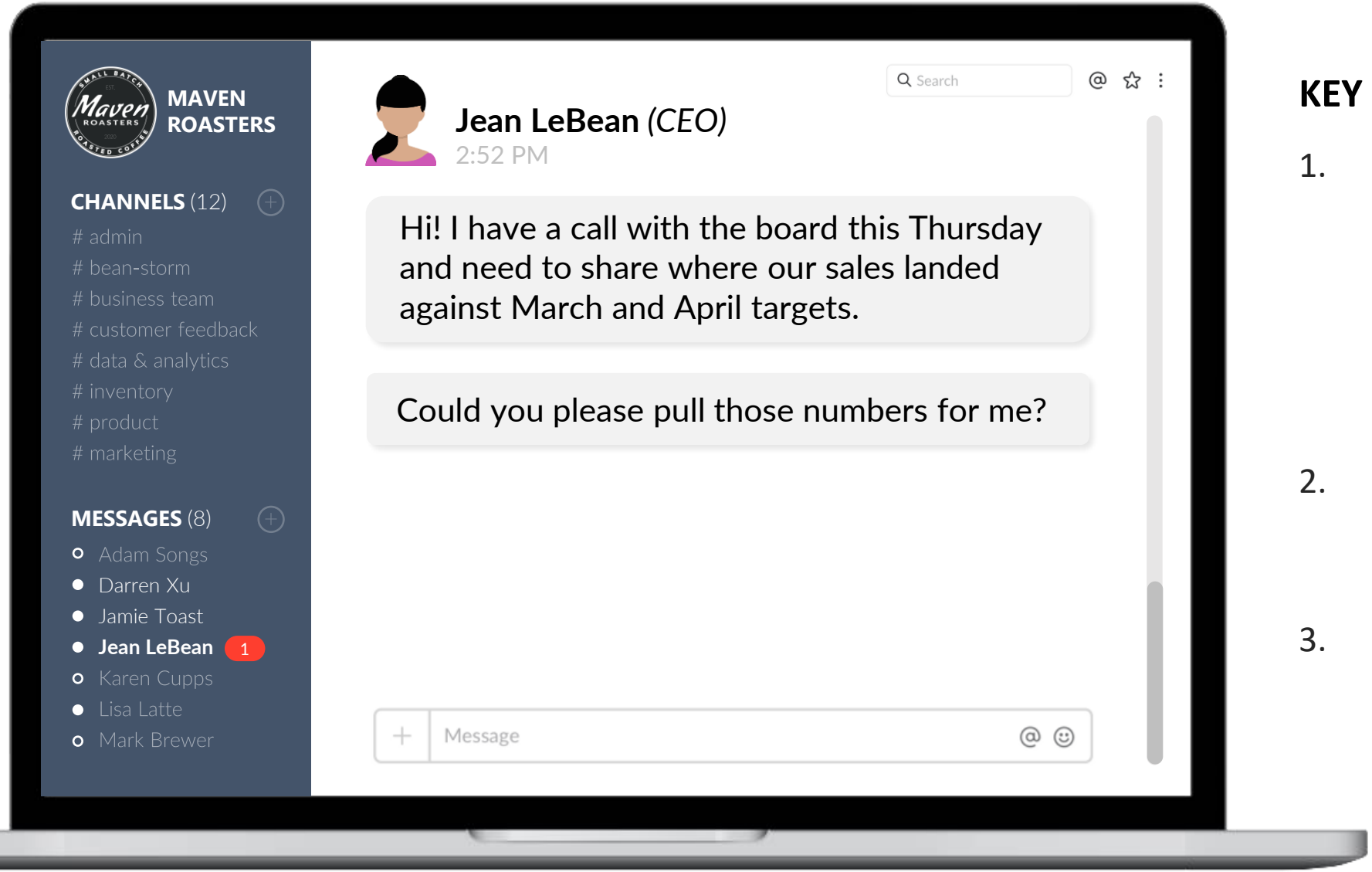
In this case, we can't create physical relationships to **Target Sales** since it's at a *different level of granularity* than the other tables in the model

***Target Sales** is at the **Month** & **Year** level*

*Both **Sales by Store** and **Calendar** are at the **Date** (daily) level*

TREATAS allows us to create *virtual, summarized* versions of our tables to match the granularity that we need to form a valid relationship

ASSIGNMENT: SALES TARGETS (TREATAS)



KEY OBJECTIVES:

1. Based on the Target Sales Union table, use **TREATAS** and create measures for Bean/Tea, Beverage, Merchandise & Food sales goals

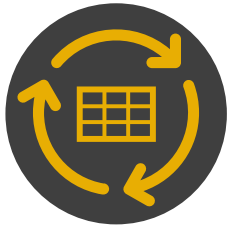
   Bonus: *Use SUMMARIZE with TREATAS*

2. Create **% of goal** measures that compare quantity sold to the goal amount

3. Add the above measures to a matrix broken down by store & target months

ITERATOR FUNCTIONS

ITERATORS



Iterator functions allow you to loop through the same expression on every row of a table in order to evaluate a single scalar value *(i.e. max, min, average)* or derive a new table

TOPICS WE'LL COVER:

- Iterator Review
- Iterator Cardinality
- CONCATENATEX
- AVERAGEX
- RANKX

COMMON USE CASES:

- *Aggregating a column into a single value (i.e. average customer age, max product price, count of orders, etc.)*
- *Returning a table of data (i.e. ADDCOLUMNS and SELECTCOLUMNS)*

REVIEW: BASIC ITERATORS

- Iterator Review
- Iterator Cardinality
- CONCATENATEX
- AVERAGEX
- RANKX

SUMX()

Returns the sum of an expression evaluated for each row in a table

=SUMX(Table, Expression)

Aggregation to apply to calculated rows

***Examples**:*

- *SUMX*
- *COUNTX*
- *AVERAGEX*
- *RANKX*
- *MAXX/MINX*

Table in which the expression will be evaluated

***Examples**:*

- *'Sales by Store'*
- *FILTER(Sales, RELATED(*

  'Products'[Category])="Clothing")

Expression to be evaluated for each row of the given table

***Examples**:*

- *[Total Orders]*
- *'Sales'[RetailPrice] * 'Sales'[Quantity]*

PRO TIP:

*Imagine the function **adding a temporary new column** to the table, calculating the value in each row (based on the expression) and then applying the aggregation to that new column*

ITERATOR CARDINALITY

- Iterator Review
- Iterator Cardinality
- CONCATENATEX
- AVERAGEX
- RANKX

Iterator cardinality is the number of rows in the table(s) being iterated; the more unique rows, the *higher* the cardinality *(this is different from **relationship** cardinality)*

Sales by Store

Index	Date	Price	Product ID	Quantity
1	1/1/2017	$18	1	2
2	1/1/2017	$18	2	6
3	1/1/2017	$14.75	3	1
…	…	…	…	…
912,422	4/30/2019	$10	10	1
912,423	4/30/2019	$3.75	79	2
912,424	4/30/2019	$3.75	36	1

When iterators reference a single table, cardinality is simply the number of unique rows *(in this case **912,424**)*

When using nested iterators, cardinality depends on whether you are using **physical** or **virtual** relationships:

- For **physical relationships**, cardinality is defined as the max number of unique rows in the *largest* table
- For **virtual relationships**, cardinality is defined as the number of unique rows in each table *multiplied together*

HEY THIS IS IMPORTANT!

*When using nested iterators, only the **innermost** "X" function can be optimized by the DAX engine. Nested iterators aren't always a bad thing, but they can have significant performance implications*

ITERATOR CARDINALITY (CONT.)

- Iterator Review
- **Iterator Cardinality**
- CONCATENATEX
- AVERAGEX
- RANKX

```
1  Customer Sales 2 =
2  SUMX(
3      'Product Lookup',
4      SUMX(
5          RELATEDTABLE( 'Sales by Store'),
6          'Sales by Store'[Unit_Price] * 'Sales by Store'[Quantity_Sold]
7      )
8  )
```

Physical relationship

- Relationship between 'Product Lookup' and 'Sales by Store'
- 'Product Lookup' contains **88 rows**
- 'Sales by Store' contains **907,841 rows**

Cardinality = 907,841

```
1  Customer Sales 3 =
2  SUMX(
3      VALUES('Product Lookup'),
4      SUMX(
5          'Sales by Store',
6          IF(
7              'Product Lookup'[product_id] = 'Sales by Store'[Product_ID],
8              'Sales by Store'[Quantity_Sold] * 'Sales by Store'[Unit_Price],
9              "-"
10         )
11     )
12 )
```

Virtual relationship

- No physical relationship between 'Product Lookup' and 'Sales by Store'
- 'Product Lookup' contains **88 rows**
- 'Sales by Store' contains **907,841 rows**

Cardinality = 79,890,008

CONCATENATEX

Iterator Review

Iterator Cardinality

CONCATENATEX

AVERAGEX

RANKX

CONCATENATEX()

Evaluates an expression for each row of the table and returns the concatenation of those values in a single string, separated by a delimiter

=CONCATENATEX(Table, Expression, [Delimiter], [OrderBy_Expression], [Order])

Table or table expression that contains the rows you want to return

***Examples**:*

- *'Product Lookup'*
- ***CONCATENATEX**(*
  VALUES('Employee Lookup')…

Column that contains values to concatenate or an expression that returns a value

***Examples**:*

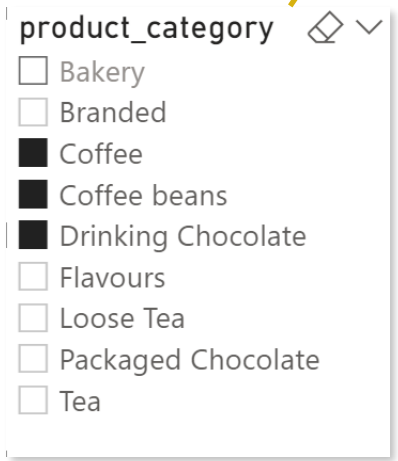
- *'Product'[Category]*
- *'Employee'[Name]*
- *[Customer Sales]*
- *7*

Optional arguments:

- ***Delimiter**: Used with concatenated expression*
  - ***Examples**: "," "&" "-" ";" etc.*
- ***OrderBy Expression**: Expression used to sort the table*
  - ***Examples**: Product Lookup[Product Category]*
- ***Order**: Order of results applied*
  - ***Examples**: ASC, DESC*

PRO TIP: ADDING DYNAMIC LABELS

- Iterator Review
- Iterator Cardinality
- **CONCATENATEX**
- AVERAGEX
- RANKX



*Product Category slicer is filtered to **Coffee, Coffee Beans & Drinking Chocolate***

```
1  Selected Product Category (CONCATENATEX) =
2  "Showing Sales For: " &
3  CONCATENATEX(
4      VALUES(
5          'Product Lookup'[product_category]
6      ),
7      'Product Lookup'[product_category],
8      ", ",
9      'Product Lookup'[product_category],
10     ASC
11 )
```

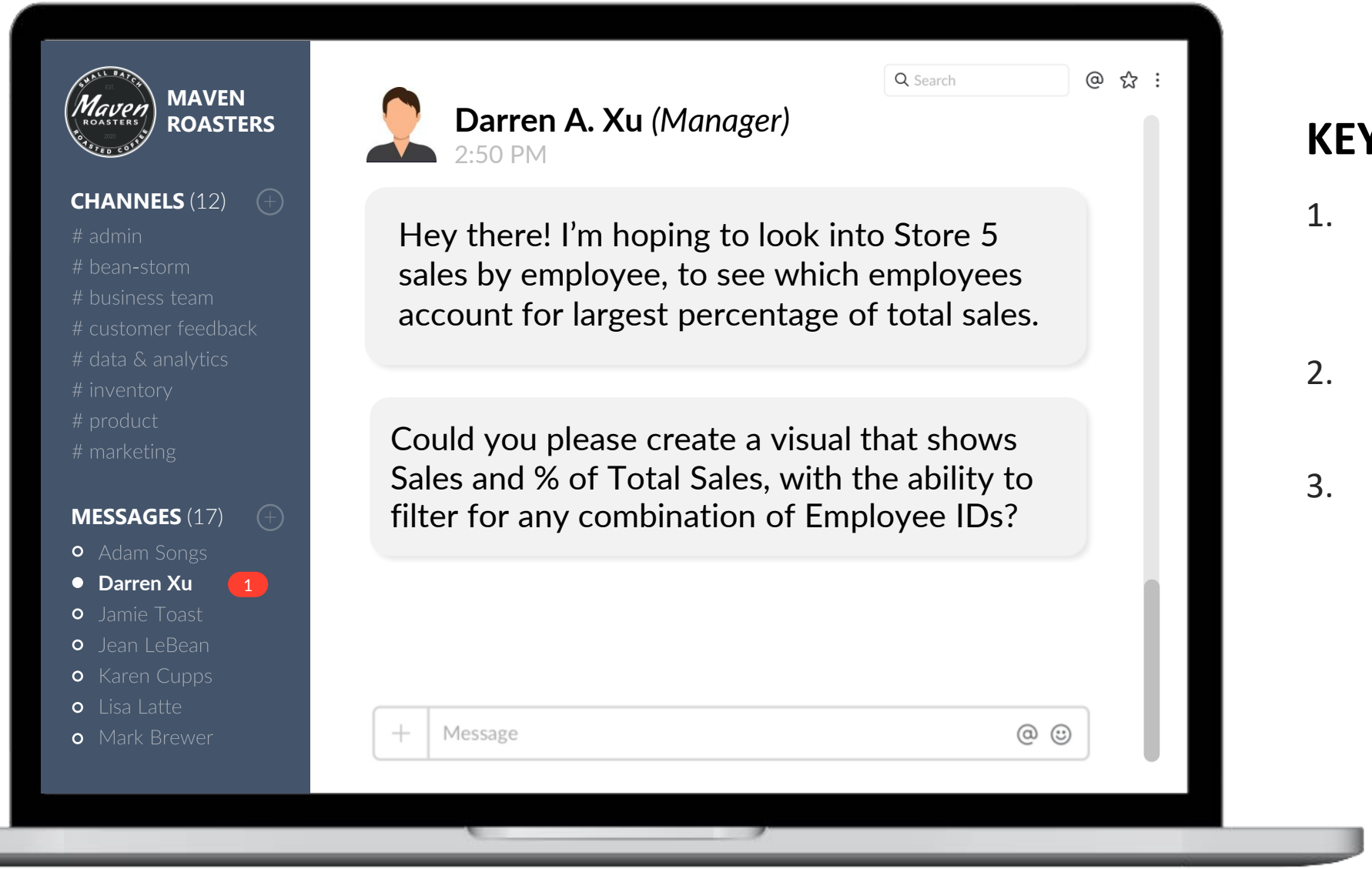
*The **Selected Product Category** measure uses **CONCATENATEX** to capture the selections from the slicer...*

store_id	Customer Sales
3	**$732,308.45**
Astoria	$732,308.45
5	**$713,956.00**
Lower Manhattan	$713,956.00
8	**$758,641.75**
Hell's Kitchen	$758,641.75
Total	**$2,204,906.20**

Showing Sales For: Coffee, Coffee beans, Drinking Chocolate

*...which can be displayed as a **dynamic label** using a card to show selected items*

ASSIGNMENT: CONCATENATEX



KEY OBJECTIVES:

1. Create a visual for Store 5 that shows **customer sales** by **store** and **employee id**
2. Create a measure to show **% of Total Sales** for Store 5
3. Use **CONCATENATEX** to define a measure that shows the employee name(s) selected and the % of Total Sales for Store 5

AVERAGEX

- Iterator Review
- Iterator Cardinality
- CONCATENATEX
- AVERAGEX
- RANKX

AVERAGEX()

Calculates the average (arithmetic mean) of a set of expressions evaluated over a table

=AVERAGEX(Table, Expression)

Table, or table expression, that contains the rows to evaluate

***Examples**:*

- *'Calendar'*
- *'Product Lookup'*

The expression that you want to evaluate

***Examples**:*

- *[Customer Sales]*
- *SUM[quantity_sold]*

HEY THIS IS IMPORTANT!

AVERAGE & AVERAGEX do **NOT** count days with zero sales when computing an average. To evaluate an average over a date range that includes dates with no sales, use **DIVIDE** & **COUNTROWS** instead

PRO TIP: MOVING AVERAGES

- Iterator Review
- Iterator Cardinality
- CONCATENATEX
- **AVERAGEX**
- RANKX

```
1  AVERAGEX Sales =
2  VAR LastTransactionDate = MAX('Calendar'[Transaction_Date])
3  VAR AverageDays = 30
4  VAR PeriodInVisual =
5  FILTER(
6      ALL(
7          'Calendar'[Transaction_Date]
8      ),
9      AND(
10         'Calendar'[Transaction_Date] > LastTransactionDate - AverageDays,
11         'Calendar'[Transaction_Date] <= LastTransactionDate
12     )
13 )
14 VAR Output =
15 CALCULATE(
16     AVERAGEX(
17         'Calendar',
18         [Customer Sales]
19     ),
20     PeriodInVisual
21 )
22 RETURN
23 Output
```

Here we're using **MAX**, **FILTER** & **ALL** to create a 30-day rolling time period *(based on the latest transaction date)*

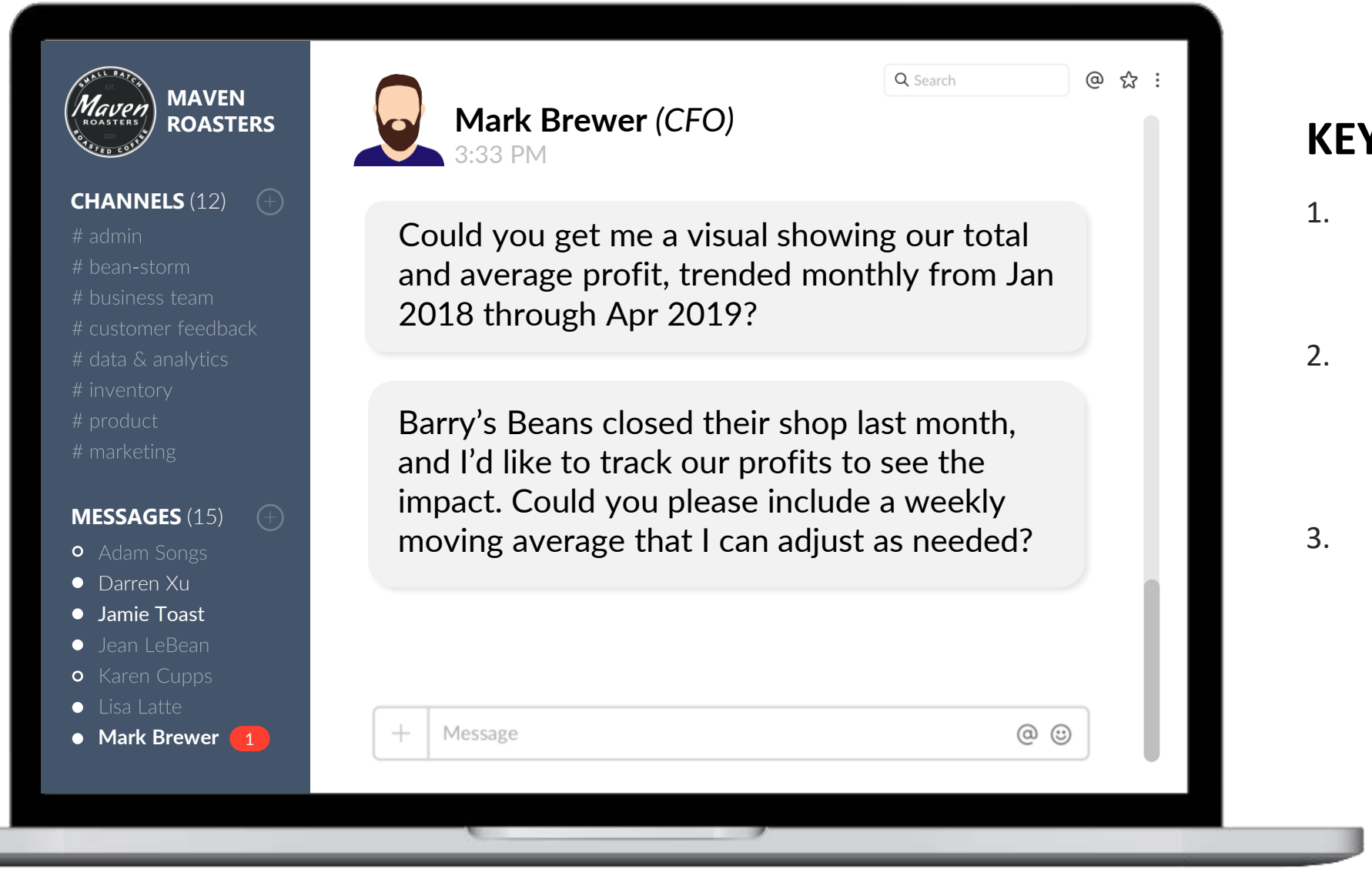
Once the rolling window is defined, we can use **AVERAGEX** to calculate the average sales over that period

- *Note that we could also use **SUMX** or **COUNTX** to calculate the rolling total/count*

PRO TIP:

*Use a **parameter** to create a dynamic, user-defined period!*

ASSIGNMENT: MOVING AVERAGES



KEY OBJECTIVES:

1. Create a matrix to show total and daily average profit by month, for Jan 2018 – Apr 2019
2. Use **GENERATESERIES** & **SELECTEDVALUE** to create a parameter with increments of 7 days over a 9-week period
3. Create a measure using **AVERAGEX** and the parameter you defined to calculate the moving average profit

RANKX

Iterator Review

Iterator Cardinality

CONCATENATEX

AVERAGEX

RANKX

RANKX()

Returns the ranking of a number in a list of numbers for each row in the table argument

=**RANKX**(Table, Expression, [Value], [Order], [Ties])

*Table or **DAX expression** that returns a table*

Examples:

- *ALL('Product Lookup'[Product])*

An expression that returns a scalar value, evaluated at each row of the table

***Examples**:*

- *[Customer Sales]*
- *SUM('Sales by Store'[quantity_sold])*

Optional arguments:

- ***Value**: Any DAX expression that returns a single scalar value whose rank is to be found (by default, the value in the **current row** is used)*
- ***Order**: Specifies how to rank (low-high vs. high-low)*
  - ***Examples**: ASC or DESC*
- ***Ties**: Determines how ties and following ranks are treated:*
  - ***SKIP** (default): Skips ranks after ties*
  - ***DENSE:** Shows the next rank, regardless of ties*

PRO TIP:

*If your "Total" row is showing a rank of 1, use **IF** & **HASONEVALUE** with **RANKX** to exclude it from the rank!*

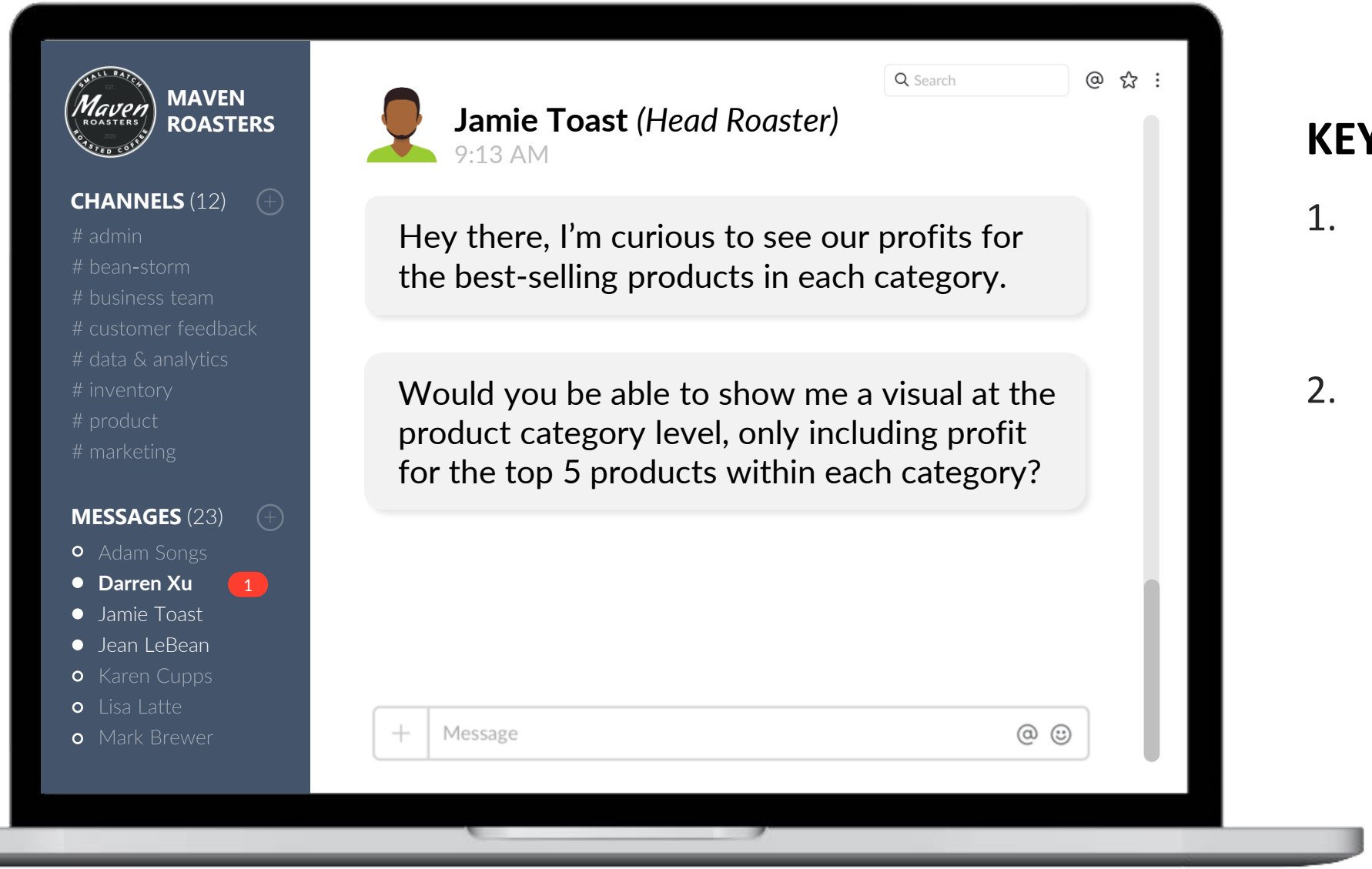
SKIP:

Rounded Customer Sales	Rank on Rounded Customer Sales
$1,700,000.00	1
$1,200,000.00	2
$500,000.00	3
$400,000.00	4
$300,000.00	5
$100,000.00	6
$100,000.00	6
$100,000.00	6
$0.00	9
$4,300,000.00	

DENSE:

Rounded Customer Sales	Rank on Rounded Customer Sales
$1,700,000.00	1
$1,200,000.00	2
$500,000.00	3
$400,000.00	4
$300,000.00	5
$100,000.00	6
$100,000.00	6
$100,000.00	6
$0.00	7
$4,300,000.00	

ASSIGNMENT: RANKX

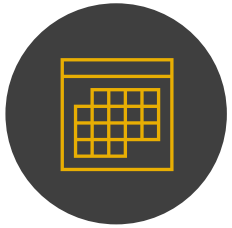


KEY OBJECTIVES:

1. Use **variables** and **RANKX** to create a single measure for Top 5 Products
2. Create a slicer for Product Category to be able to analyze the top 5 across different categories

ADVANCED TIME INTELLIGENCE

ADVANCED TIME INTELLIGENCE



DAX offers a range of powerful **time intelligence** functions, which allow you to build custom calendars, define dynamic date ranges, and compare performance across specific periods

TOPICS WE'LL COVER:

- **Automatic Date Tables**
- **Calendar Functions**
- **Date Formatting**
- **Time Intelligence Functions**
- **Week-Based Calculations**

COMMON USE CASES:

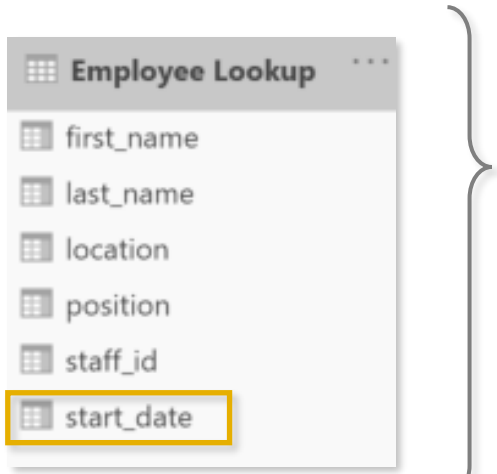
- *Building custom calendar tables (rather than using default, auto-generated versions)*
- *Calculating period-over-period, moving average, or running total calculations*
- *Managing non-traditional time periods like week-based retail calendars (5-4-5 or 5-4-4)*

AUTOMATIC DATE TABLES

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

By default, Power BI automatically creates a **hidden date table** for any table that contains a **Date** or **DateTime** column on the one-side of a relationship

- Auto-generated calendars include *all* dates through the end of the year, regardless of the actual date range in the table



Date	Day	MonthNo	Month	QuarterNo	Quarter	Year
1/01/2017 00:00:00	1	1	January	1	Q1	2017
1/02/2017 00:00:00	2	1	January	1	Q1	2017
1/03/2017 00:00:00	3	1	January	1	Q1	2017
1/04/2017 00:00:00	4	1	January	1	Q1	2017
1/05/2017 00:00:00	5	1	January	1	Q1	2017
1/06/2017 00:00:00	6	1	January	1	Q1	2017
1/07/2017 00:00:00	7	1	January	1	Q1	2017
1/08/2017 00:00:00	8	1	January	1	Q1	2017
1/09/2017 00:00:00	9	1	January	1	Q1	2017
1/10/2017 00:00:00	10	1	January	1	Q1	2017
1/11/2017 00:00:00	11	1	January	1	Q1	2017
1/12/2017 00:00:00	12	1	January	1	Q1	2017

*Automatically creates a **hidden** date table containing all these columns*

PROS & CONS: AUTOMATIC DATE TABLES

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

PROS:

- Automatically generated
- Enables (some) time intelligence functionality by default
- Simplifies data model creation and management
- Does not require an advanced understanding of DAX

CONS:

- Hidden from view, cannot be modified/customized
- Generated for every date field across every lookup/dimension table *(bloats model size)*
- Can't be enabled or disabled at the table-level
- Hierarchies aren't automatically generated *(if grouped by month, would summarize that month across ALL years)*
- Each automatic date table can *only* filter the table it corresponds to *(cannot traverse table relationships)*

PRO TIP:

*Turn OFF the **auto date/time** feature in Power BI Desktop and either import a date dimension table or create your own using **CALENDAR** functions (more on this soon!)*

DATE TABLE REQUIREMENTS

If you import or create your own date table, **it must meet these requirements:**

- ✓ Must contain *all* the days for all years represented in your fact tables
- ✓ Must have at least one field set as a ***Date*** or ***DateTime*** datatype
- ✓ Cannot contain duplicate dates or datetime values
- ✓ If using a time component within a date column, all times must be identical *(i.e. 12:00)*
- ✓ Should be marked as a **date table** *(not required but a best practice)*

HEY THIS IS IMPORTANT!

*If **Time** is present in your date field, split the time component into a new column (this adheres to relationship requirements and decreases column cardinality)*

CALENDAR

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

CALENDAR()

Returns a table with one column of all dates between start and end date

=CALENDAR(StartDate, EndDate)

***Start** and **End** dates of your calendar table. Can be explicitly assigned or defined using DAX functions (i.e. MIN/MAX)*

Examples:

- ***CALENDAR**(*
  DATE(2019,01,01), DATE(2020,12,31))
- ***CALENDAR**(*
  DATE(YEAR (MIN(Sales by Store[Transaction Date])), 1, 1),
  DATE(YEAR (MAX(Sales by Store[Transaction Date])), 12, 31)

Date
1/1/2017 12:00:00 AM
1/2/2017 12:00:00 AM
1/3/2017 12:00:00 AM
1/4/2017 12:00:00 AM
1/5/2017 12:00:00 AM
1/6/2017 12:00:00 AM
1/7/2017 12:00:00 AM
1/8/2017 12:00:00 AM
1/9/2017 12:00:00 AM
1/10/2017 12:00:00 AM
1/11/2017 12:00:00 AM
1/12/2017 12:00:00 AM
1/13/2017 12:00:00 AM
1/14/2017 12:00:00 AM
1/15/2017 12:00:00 AM
1/16/2017 12:00:00 AM
1/17/2017 12:00:00 AM

CALENDARAUTO

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

CALENDARAUTO()

Returns a table with one column of dates based on a fiscal year end month. The Range of dates is calculated automatically based on data in the model

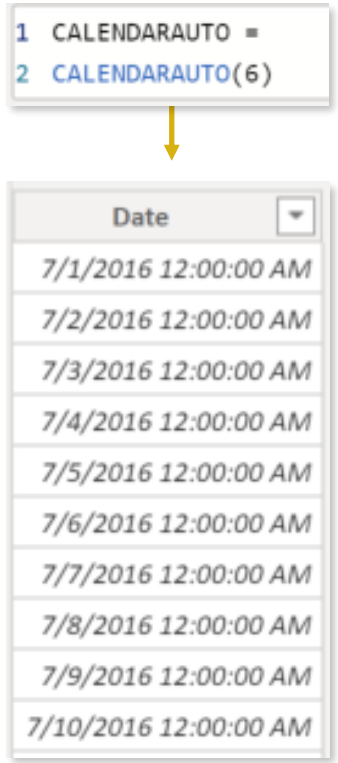
=**CALENDARAUTO**(FiscalYearEndMonth)

*An integer from **1** to **12** that represents the last month of the fiscal year.*
Can be explicitly assigned or created using DAX functions (i.e. MIN/MAX)

***Examples**:*

- ***CALENDARAUTO**(6)*
- *Calendar Table =*

```
VAR MinYear = YEAR( MIN( 'Sales by Store'[Transaction Date]))
VAR MaxYear= YEAR( MAX( 'Sales by Store'[Transaction Date]))
RETURN
    FILTER( CALENDARAUTO(),
        YEAR( [Date] ) >= MinYear &&
        YEAR( [Date] ) <= MaxYear )
```



PRO TIP: BUILDING A REUSABLE DATE TABLE

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

```
CALENDARAUTO Date Table =
VAR MinYear = YEAR(MIN('Sales by Store'[transaction_date]))
VAR MaxYear = YEAR(MAX('Sales by Store'[transaction_date]))

RETURN
ADDCOLUMNS(
    FILTER(
        CALENDARAUTO(),
        YEAR( [Date] ) >= MinYear &&
        YEAR( [Date] ) <= MaxYear
    ),
    "Year", YEAR([Date]),
    "Quarter Number", INT(FORMAT([Date],"q")),
    "Quarter", "Q" & INT(FORMAT([Date], "q")),
    "Month Number", MONTH([Date]),
    "Month Short", FORMAT([Date], "mmm"),
    "Week Number", WEEKNUM([Date])
)
```

This code can be repurposed and recycled across multiple data models!

Date	Year	Quarter Number	Quarter	Month Number	Month Short	Week Number
1/1/2017 12:00:00 AM	2017	1	Q1	1	Jan	1
1/2/2017 12:00:00 AM	2017	1	Q1	1	Jan	1
1/3/2017 12:00:00 AM	2017	1	Q1	1	Jan	1
1/4/2017 12:00:00 AM	2017	1	Q1	1	Jan	1
1/5/2017 12:00:00 AM	2017	1	Q1	1	Jan	1
1/6/2017 12:00:00 AM	2017	1	Q1	1	Jan	1
1/7/2017 12:00:00 AM	2017	1	Q1	1	Jan	1
1/8/2017 12:00:00 AM	2017	1	Q1	1	Jan	2
1/9/2017 12:00:00 AM	2017	1	Q1	1	Jan	2
1/10/2017 12:00:00 AM	2017	1	Q1	1	Jan	2
1/11/2017 12:00:00 AM	2017	1	Q1	1	Jan	2
1/12/2017 12:00:00 AM	2017	1	Q1	1	Jan	2

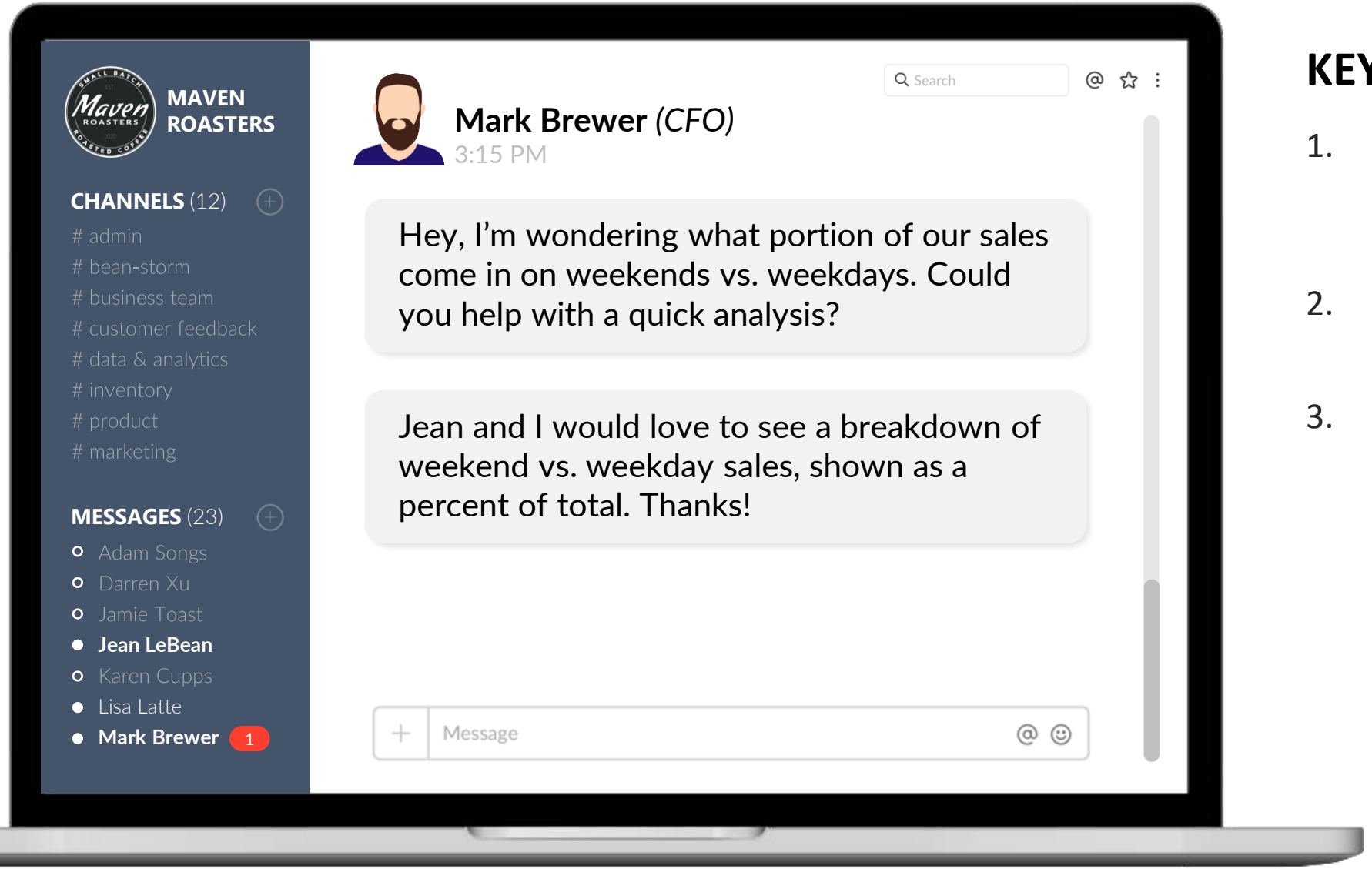
DATE FORMATTING

Use the **FORMAT** function to specify date/time formatting. Common examples include:

Code	Description	Example
dddd	Display the **full-day** name	Sunday, Monday, etc.
ddd	Display the **short-day** name	Sun, Mon, Tue, etc.
mmmm	Show full **month** name	July, August, etc.
mmm	Show abbreviated **month** name	Jul, Aug, Sept, etc.
mm	Display the **month** as a two-digit number	08, 09, 10, 11, etc.
q	Display the **quarter** of the year	1, 2, 3, 4
yyyy	Show the year as a four-digit number	2018, 2019, 2020, etc.

**Head to https://docs.microsoft.com/en-us/dax/custom-date-and-time-formats-for-the-format-function for more information about custom date and time formats and a complete list of user-defined date/time formats*

ASSIGNMENT: DATE FORMATTING



KEY OBJECTIVES:

1. Add columns to the calendar table to capture the weekday number and name
2. Add a binary Y/N column to flag weekend dates
3. Use a Matrix visual to show the percent of total sales for weekdays vs weekends

COMMON TIME INTELLIGENCE FUNCTIONS

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- **Time Intelligence Functions**
- Week-Based Calculations

Time Intelligence functions allow you to define and compare custom time periods:

Performance-to-Date

Functions commonly used to calculate performance through the current date

Common Examples:

- DATESYTD
- DATESQTD
- DATESMTD

Time Period Shift

Functions commonly used to compare performance between specific periods

Common Examples:

- SAMEPERIODLASTYEAR
- DATEADD
- PARALLELPERIOD
- PREVIOUSYEAR *(QUARTER/MONTH/DAY)*
- NEXTYEAR *(QUARTER/MONTH/DAY)*

Running Total

Functions commonly used to calculate running totals or moving averages

Common Examples:

- DATESINPERIOD

REVIEW: COMMON PATTERNS

Use these common **time intelligence patterns** for YTD, period-over-period, or running total calculations:

Performance To-Date

```
= CALCULATE(Measure, DATESYTD(Calendar[Date]))
```

Use **DATESQTD** *for Quarters or* **DATESMTD** *for Months*

Previous Period

```
= CALCULATE(Measure, DATEADD(Calendar[Date], -1, MONTH))
```

*Select an interval (***DAY***,* **MONTH***,* **QUARTER***, or* **YEAR***) and the # of intervals to compare (i.e. previous month, rolling 10-day)*

Running Total

```
= CALCULATE(Measure,
        DATESINPERIOD(Calendar[Date], MAX(Calendar[Date]), -10, DAY))
```

PRO TIP:

To calculate a ***moving average****, use a running total calculation and* ***divide by the number of intervals***

PARALLELPERIOD

Automatic Date Tables

Calendar Functions

Date Formatting

Time Intelligence Functions

Week-Based Calculations

PARALLELPERIOD()

Returns a column of dates from a parallel period, by shifting the dates specified forward or backward in time based on a given interval (month/quarter/year)

=PARALLELPERIOD(Date, NumberOfIntervals, Interval)

The name of a column containing dates or a one column table containing dates

Example:

- *Calendar[Transaction_Date]*

Number of Intervals (positive or negative); If a decimal is supplied, number is rounded to nearest whole integer

Example:

- *1, 2, 3, etc.*
- *-1,-2,-3 etc.*
- *-1.4 (-1) -1.5 (-2)*

*Interval value **can only** be:*

- ***Month***
- ***Quarter***
- ***Year***

HEY THIS IS IMPORTANT!

PARALLELPERIOD computes the *entire* period in the interval *(i.e. entire year, quarter, etc.)*. Values in **total** rows may not reflect the expected total if partial periods are present!

PARALLELPERIOD EXAMPLE

- Automatic Date Tables
- **Calendar Functions**
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

```
1  Last Quarter's Sales (PARALLELPERIOD) =
2  CALCULATE(
3      [Customer Sales],
4      PARALLELPERIOD(
5          'Calendar'[Transaction_Date],
6          -1,
7          QUARTER
8      )
9  )
```

Year_ID	Customer Sales	Last Quarter's Sales (PARALLELPERIOD)
2017	**$1,678,074.11**	**1,158,966.35**
January	$81,845.09	
February	$76,273.99	
March	$99,154.43	
April	$119,309.01	257,273.51
May	$157,208.99	257,273.51
June	$166,899.78	257,273.51
July	$157,968.55	443,417.78
August	$154,485.32	443,417.78
September	$145,821.19	443,417.78
October	$169,223.54	458,275.06
November	$179,999.30	458,275.06
December	$169,884.92	458,275.06
2018	**$1,916,544.75**	**1,901,863.54**
January	$141,284.63	519,107.76
February	$124,030.84	519,107.76
March	$143,475.17	519,107.76
April	$149,780.08	408,790.64
May	$173,257.84	408,790.64
June	$174,349.34	408,790.64
July	$164,985.34	497,387.26
August	$161,108.60	497,387.26
September	$150,483.94	497,387.26
October	$174,293.02	476,577.88
November	$184,401.31	476,577.88
December	$175,094.64	476,577.88
2019	**$658,086.02**	**959,528.19**
January	$146,863.61	533,788.97
February	$129,473.65	533,788.97
March	$149,401.96	533,788.97
April	$232,346.80	425,739.22
Total	**$4,252,704.88**	**4,020,358.08**

*Total returned for the **entire previous quarter** on each row*

*Year-level total = **Q4 2018 + Q1 2019***

*Overall total is "missing" **April 2019***

PREVIOUSQUARTER

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

PREVIOUSQUARTER

Returns a table containing a column of all dates from the previous quarter, based on the first date in the date range specified

=**PREVIOUSQUARTER**(Dates)

The name of a column containing dates or a one column table containing dates

Example:

- *Calendar[Transaction_Date]*

HEY THIS IS IMPORTANT!

PREVIOUSQUARTER works similarly to **PARALLELPERIOD** *(QUARTER) and* computes the *entire* prior period (but handles totals differently)

Year_ID	Customer Sales	Last Quarter's Sales (PREVIOUSQUARTER)
2017	**$1,258,224.88**	**$257,273.51**
May	$113,942.28	$257,273.51
June	$166,899.78	$257,273.51
July	$157,968.55	$443,417.78
August	$154,485.32	$443,417.78
September	$145,821.19	$443,417.78
October	$169,223.54	$458,275.06
November	$179,999.30	$458,275.06
December	$169,884.92	$458,275.06
2018	**$1,916,544.75**	**$519,107.76**
January	$141,284.63	$519,107.76
February	$124,030.84	$519,107.76
March	$143,475.17	$519,107.76
April	$149,780.08	$408,790.64
May	$173,257.84	$408,790.64
June	$174,349.34	$408,790.64
July	$164,985.34	$497,387.26
August	$161,108.60	$497,387.26
September	$150,483.94	$497,387.26
October	$174,293.02	$476,577.88
November	$184,401.31	$476,577.88
December	$175,094.64	$476,577.88
2019	**$658,086.02**	**$533,788.97**
January	$146,863.61	$533,788.97
February	$129,473.65	$533,788.97
March	$149,401.96	$533,788.97
April	$232,346.80	$425,739.22
Total	**$3,832,855.65**	**$257,273.51**

Transaction_Date
is after 5/9/2017

*When first date specified is 5/9/2017, totals for May and June reflect **Jan-Mar (Q1) 2017***

*2018 Total row reflects the total for **Q4 2017***

*With no date context, total is based on first visible date in the table (**5/9/2017**)*

SAMEPERIODLASTYEAR

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

SAMEPERIODLASTYEAR()

Returns a set of dates in the current selection from the previous year

= **SAMEPERIODLASTYEAR**(Dates)

The name of a column containing dates or a one column table containing dates

Example:

- *Calendar[Transaction_Date]*

HEY THIS IS IMPORTANT!

Pay attention to totals and filter context! **SAMEPERIODLASTYEAR** is equivalent to *DATEADD('Dates', -1, YEAR)* and may return values which aren't always intuitive or straightforward to understand

Year_ID	YTD Sales	Last Year's Sales (SAMEPERIODLASTYEAR)
2018	**$1,916,544.75**	**$1,445,900.03**
March	$408,790.64	$25,099.43
April	$558,570.72	$119,309.01
May	$731,828.56	$157,208.99
June	$906,177.90	$166,899.78
July	$1,071,163.24	$157,968.55
August	$1,232,271.84	$154,485.32
September	$1,382,755.78	$145,821.19
October	$1,557,048.80	$169,223.54
November	$1,741,450.11	$179,999.30
December	$1,916,544.75	$169,884.92
2019	**$658,086.02**	**$558,570.72**
January	$146,863.61	$141,284.63
February	$276,337.26	$124,030.84
March	$425,739.22	$143,475.17
April	$658,086.02	$149,780.08
Total	**$658,086.02**	**$2,004,470.75**

*Total Sales for **3/24/2017 – 3/31/2017***

*2019 total has filter context of **1/1/2019 – 4/30/2019**, and returns **2018 YTD total through April***

SYNTAX SUGAR: SAMEPERIODLASTYEAR

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- **Time Intelligence Functions**
- Week-Based Calculations

SAMEPERIODLASTYEAR is really just a shortcut for **DATEADD** and **YEAR**:

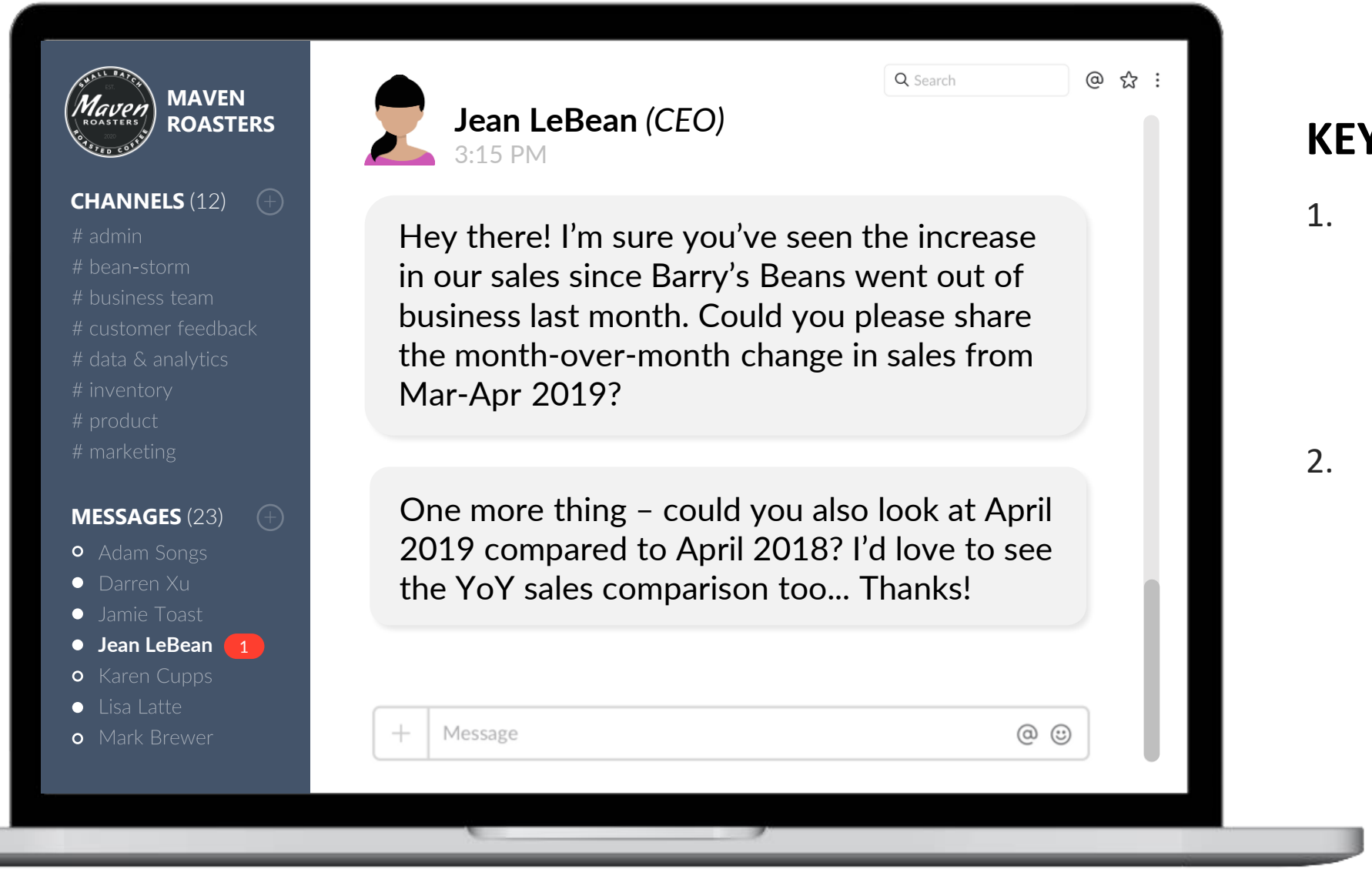
How it's written:

```
1 Store 3 Sales – Last Year =
2 CALCULATE(
3     [Total Sales],
4     SAMEPERIODLASTYEAR('Calendar'[Transaction_Date])
5 )
```

How it's interpreted:

```
1 Store 3 Sales – Last Year =
2 CALCULATE(
3     [Total Sales],
4     DATEADD(
5             'Calendar'[Transaction_Date],
6             -1,
7             YEAR)
8 )
```

ASSIGNMENT: TIME PERIODS



KEY OBJECTIVES:

1. Using time intelligence functions to calculate total sales for the previous month, as well as the percent change month-over-month
2. Calculate the year-over-year change in sales from April 2018 vs. April 2019

WEEK-BASED CALCULATIONS

Week-based DAX calculations *(i.e. previous week sales, same week last year)* can be especially challenging for several reasons:

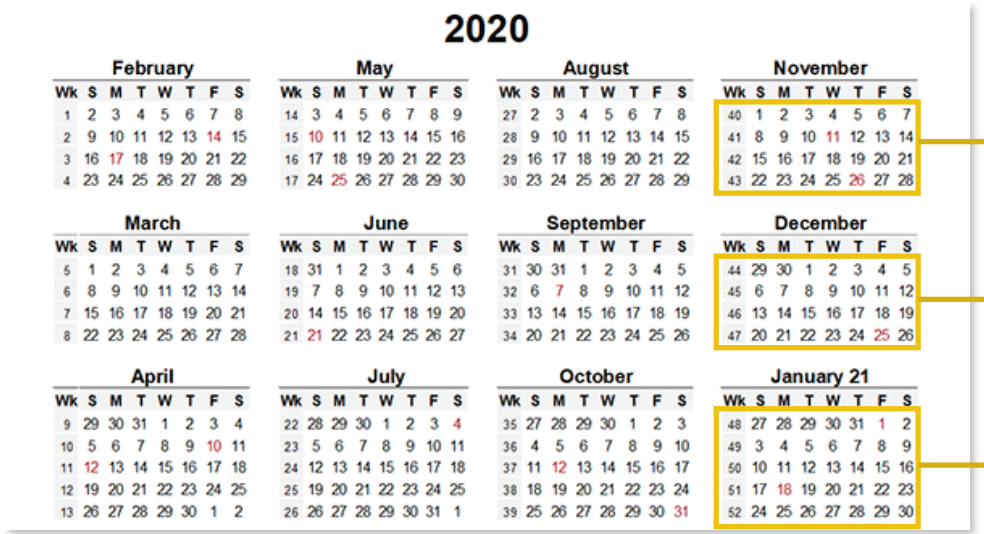
- Can't use standard DAX functions *(i.e. PARALLELPERIOD, SAMEPERIODLASTYEAR)*
- Weeks can start on various days *(Sunday, Monday, etc.)*
- There can be partial weeks in any given month, quarter or year *(i.e. partial 53rd week)*
- Weeks can be grouped differently based on different fiscal calendars *(i.e. 5-4-4, 4-5-4)*

PRO TIP:

*Using a **5-4-4** or **4-5-4** fiscal calendar (common in the retail industry) can enable some week-based calculations, but you still can't use standard DAX time intelligence functions*

4-4-5 & 4-5-4 FISCAL CALENDARS

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- **Week-Based Calculations**



PROS:

- Standardizes the number of days in each week **(7)**
- Standardizes the number of weeks in each year **(52)**

CONS:

- Does not have standard start & end dates for years, quarters, or months
- Cannot be used with standard DAX time intelligence functions

PREVIOUS FISCAL WEEK (4-5-4)

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

To calculate performance for the **previous fiscal week**, you can use **DATEADD** with an interval of **-7 days**:

```
1 Last Week's Sales 4-5-4 (DATEADD) =
2 CALCULATE(
3     [Customer Sales],
4     DATEADD(
5         '4-5-4 Calendar'[Date],
6         -7,
7         DAY
8     )
9 )
```

*This works with 4-5-4 or 4-4-5 fiscal calendars, but can break down when applied to standard calendars where years often contain **partial weeks***

PRO TIP:

*Use **DATEADD** with an interval of **-364 days** to compare the same week last year!*

FISCAL PERIOD TO DATE (4-5-4)

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- **Week-Based Calculations**

Instead of a standard DAX **performance-to-date** functions *(i.e. DATESYTD, DATESMTD)*, you can use a combination of **CALCULATE**, **MAX**, and **HASONEVALUE**:

*Here we're calculating **QTD sales** based on a 4-5-4 fiscal calendar*

```
1  QTD Sales (4-5-4) =
2  VAR MaxDate = MAX('4-5-4 Calendar'[Date])
3  VAR MaxPeriod = MAX('4-5-4 Calendar'[FiscalQuarterYear])
4  VAR Output =
5  IF(
6      HASONEVALUE(
7          '4-5-4 Calendar'[FiscalQuarter]
8      ),
9      CALCULATE(
10         [Customer Sales],
11         '4-5-4 Calendar'[Date] <= MaxDate,
12         '4-5-4 Calendar'[FiscalQuarterYear] = MaxPeriod
13     ),
14     "-"
15 )
16 RETURN
17 Output
```

*You can replace **FiscalQuarter** with the fiscal year or fiscal month for **YTD** and **MTD** calculations*

FISCAL PREVIOUS PERIOD (4-5-4)

- Automatic Date Tables
- Calendar Functions
- Date Formatting
- Time Intelligence Functions
- Week-Based Calculations

Instead of a standard DAX **time period shifting** functions *(i.e. PREVIOUSYEAR, PARALLELPERIOD)*, you can use **CALCULATE**, **FILTER**, **ALL** and **SELECTEDVALUE**:

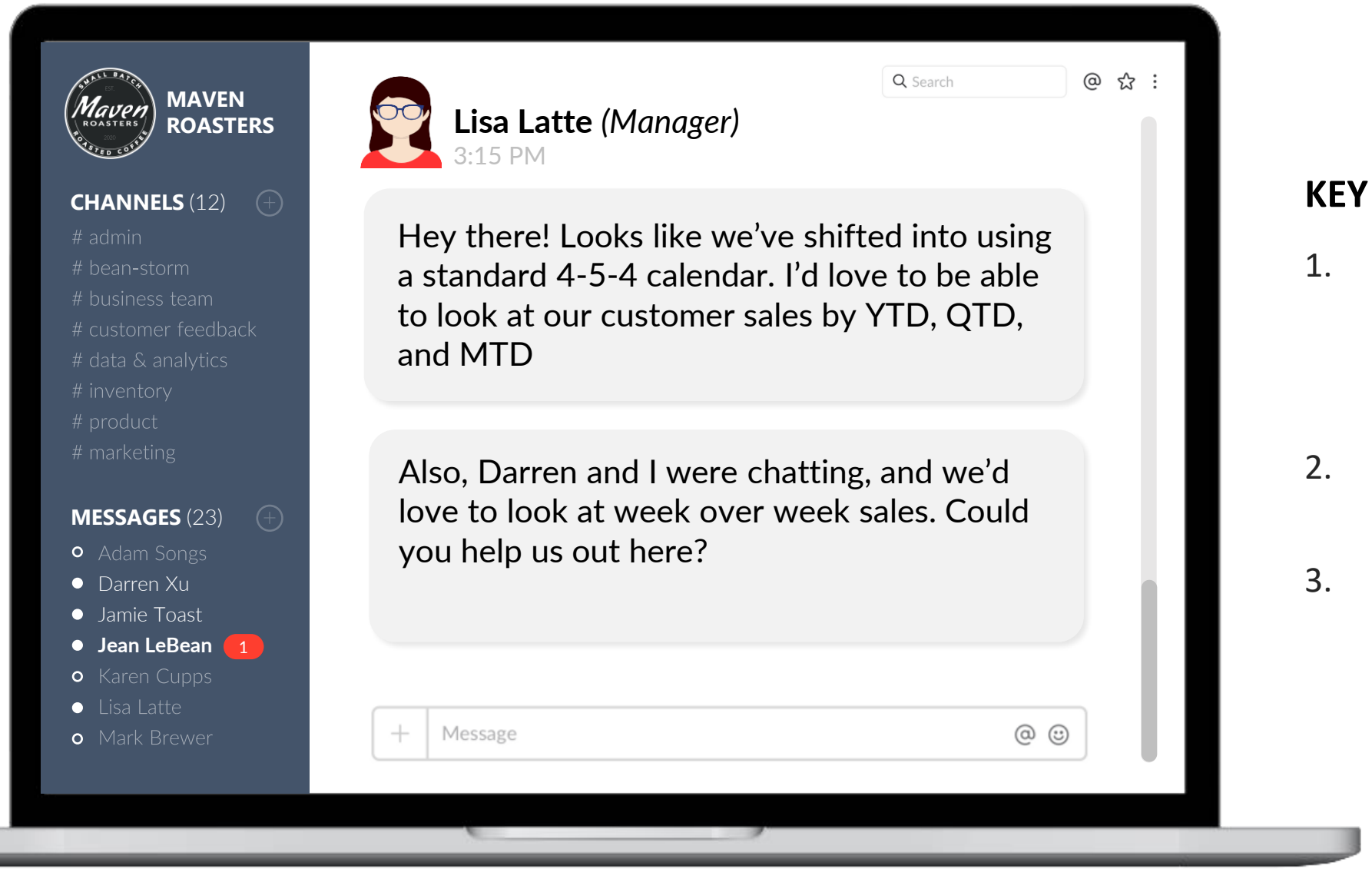
*Here we're calculating **QTD sales for the previous quarter** based on a 4-5-4 fiscal calendar*

```
1  Last QTD Sales (4-5-4) =
2  VAR LastPeriod =
3  CALCULATE(
4      [Customer Sales],
5      FILTER(
6          ALL(
7              '4-5-4 Calendar'
8          ),
9          IF(
10             SELECTEDVALUE('4-5-4 Calendar'[FiscalQuarter]) = 1,
11             '4-5-4 Calendar'[FiscalQuarter] = 4 &&
12             '4-5-4 Calendar'[FiscalYear] = SELECTEDVALUE('4-5-4 Calendar'[FiscalYear]) -1,
13             '4-5-4 Calendar'[FiscalYear] = SELECTEDVALUE('4-5-4 Calendar'[FiscalYear]) &&
14             '4-5-4 Calendar'[FiscalQuarter] = SELECTEDVALUE('4-5-4 Calendar'[FiscalQuarter]) -1
15         )
16     )
17 )
18 RETURN
19 LastPeriod
```

*You can replace **FiscalQuarter** with the fiscal month or fiscal week for **MTD** and **WTD** calculations*

*Just make sure to update the fiscal period to either **4**, **12** or **52**!*

ASSIGNMENT: 4-5-4 CALENDAR

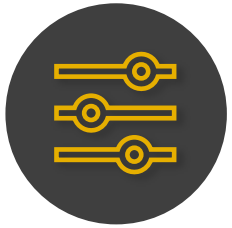


KEY OBJECTIVES:

1. Using the 4-5-4 Calendar and **[Customer Sales]** create measures for YTD and MTD sales
2. Create a measure to compute week-over-week % change
3. Create a matrix (or two) to visualize your results

PERFORMANCE TUNING

SNEAK PEEK: PERFORMANCE TUNING



Tools like **DAX Studio** and Power BI's **Performance Analyzer** can you help you troubleshoot issues, measure load times for visuals/DAX queries, and optimize your code

TOPICS WE'LL COVER:

- **Performance Analyzer**
- **Copy Query**
- **DAX Studio**
- **Optimization Workflow**

COMMON USE CASES:

- *Identify issues with slow-loading visuals or queries*
- *Test and compare measures to determine the impact on speed and performance*
- *Optimize measures or report configurations to minimize processing and load times*

PERFORMANCE ANALYZER

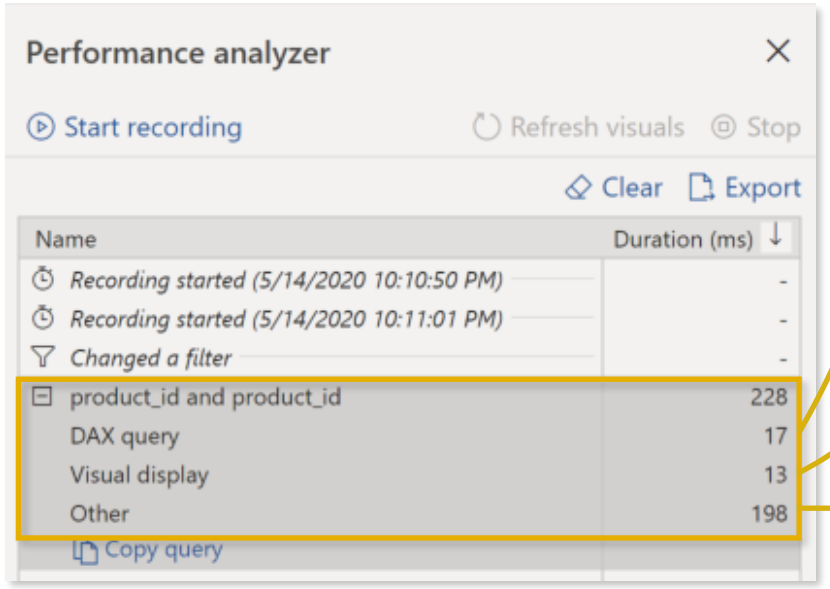
Performance Analyzer

Copy Query

DAX Studio

Optimization Workflow

Power BI Desktop's **Performance Analyzer** records user actions *(like Excel's macro recorder)*, and tracks the load time *(in milliseconds)* for each step in the process:



DAX Query

- *Shows the amount of time it takes for the visual to send the query to the engines, and for the engines to return the result (**Note:** DAX Studio can only help optimize this)*

Visual Display

- *Shows the amount of time it takes for the visual to populate, or "draw", on the screen. Includes time to retrieve web-based and geocoded images*

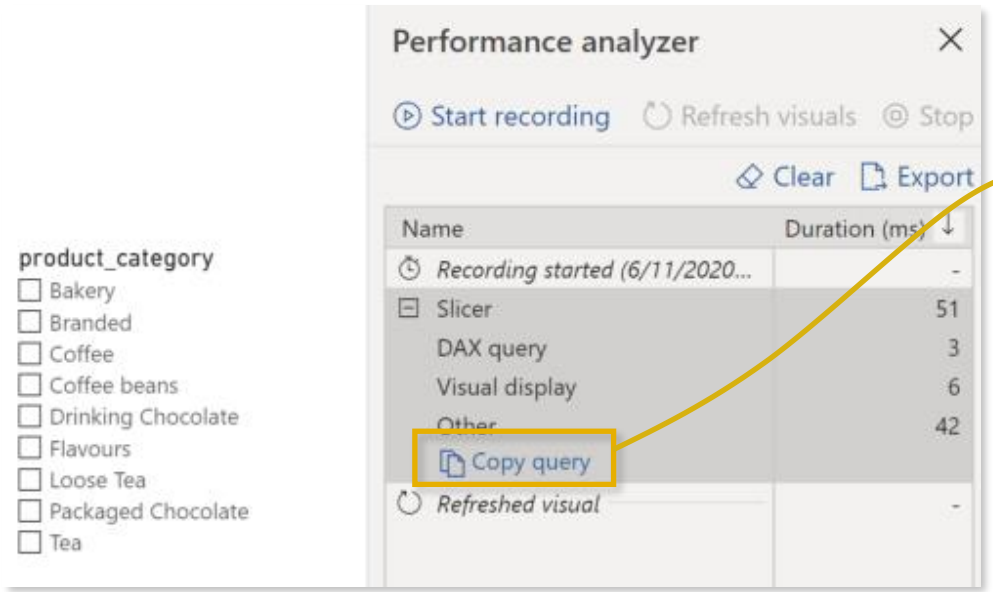
Other

- *Shows the amount of time required by the visual to prepare the query, wait for other visuals to complete their queries and perform other processing tasks*

PRO TIP: COPY QUERY

- Performance Analyzer
- **Copy Query**
- DAX Studio
- Optimization Workflow

The **Performance Analyzer** includes a **Copy Query** option, which you can use to copy the actual code that DAX generates behind the scenes to produce specific visuals:




```
Slicer Query = // DAX Query
EVALUATE
  TOPN(
    101,
    VALUES('Product Lookup'[product_category]),
    'Product Lookup'[product_category],
    1
  )

ORDER BY
  'Product Lookup'[product_category]
```

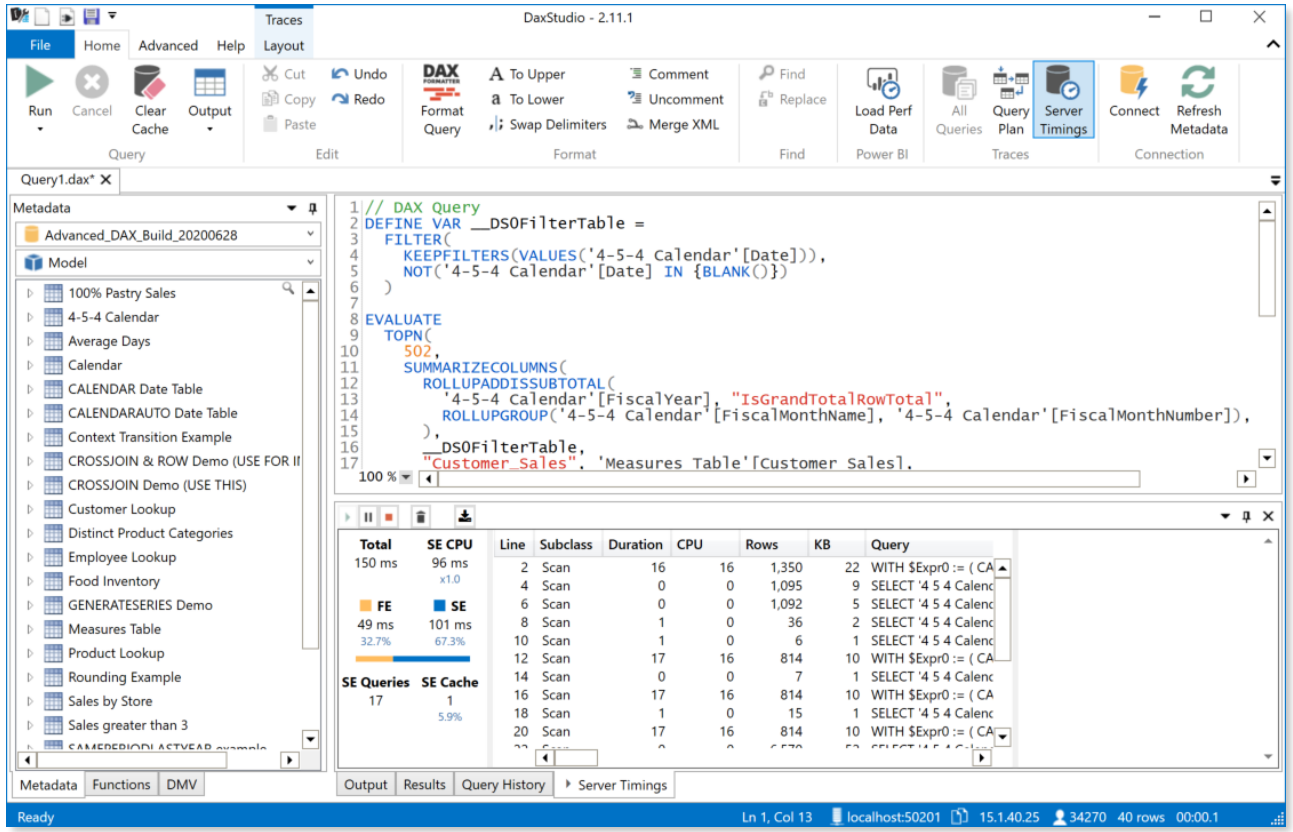
*Remember that **VALUES** will include a blank if it exists in the model? Well if you see a blank option in a slicer, this is why!*

DAX STUDIO

DAX Studio is a free tool that allows you to connect to your Power BI data model to test and optimize your DAX queries

- Performance Analyzer
- Copy Query
- **DAX Studio**
- Optimization Workflow

www.Daxstudio.org



OPTIMIZATION WORKFLOW

Performance Analyzer

Copy Query

DAX Studio

Optimization Workflow

1 Start **Performance Analyzer** in Power BI Desktop

- *Click "**Start Recording**", any actions will be logged and displayed in the Performance Analyzer pane*

2 Analyze your query **load times**

- *Each part of the query will likely take a different amount of time, and if the **DAX Query** load time is long (100-200ms or more) you can likely optimize your code using DAX Studio*
- ***Visual display** load time may be decreased by changing the visual type or reducing data points*

3 Copy & paste or import queries into **DAX Studio**

- *Copy and paste an individual query and analyze its performance in DAX Studio, or export the entire performance analyzer results and use **Analyze Performance Data** to import a file containing all queries*

4 Use DAX studio to **optimize your code**

- *DAX Studio uses some specific functions which allow you to analyze your DAX queries (i.e. **DEFINE**, **EVALUATE** and **ORDER BY**)*
- *In an optimized query, the storage engine (SE) should carry the majority of the CPU workload*

WRAPPING UP

RESOURCES & NEXT STEPS

Need to brush up on your **Power BI** skills? Make sure to complete the full Power BI Specialist stack from Maven Analytics:

- ***Up & Running with Power BI Desktop***
- ***Publishing to Power BI Service***
- ***Advanced DAX for Business Intelligence***

Remember to check out these **helpful resources** for additional support:

- ***Dax.guide*** *&* ***docs.microsoft.com*** *for DAX function reference*
- ***Mavenanalytics.io/data-playground*** *for free practice datasets*
- ***pbiusergroup.com*** *for helpful forums and local meetup groups*
- ***Microsoft Power BI*** *and* ***Guy in a Cube*** *YouTube channels for demos and advanced tutorials*

Any feedback? Please take a moment to leave a rating or review!

- *Please reach out if there's anything we can do to improve your experience – we're here to help!*


THANK YOU