DirectQuery in SQL Server 2016 Analysis Services

Technical White Paper

Published: January 2017

Applies to: Microsoft SQL Server 2016 Analysis Services, Tabular

Summary: DirectQuery transforms the Microsoft SQL Server Analysis Services Tabular model into a metadata layer on top of an external database. For SQL Server 2016, DirectQuery was redesigned for dramatically improved speed and performance, however, it is also now more complex to understand and implement. There are many tradeoffs to consider when deciding when to use DirectQuery versus in-memory mode (VertiPaq). Consider using DirectQuery if you have either a small database that is updated frequently or a large database that would not fit in memory.

Authors:

Marco Russo (SQL Server MVP and BI Consultant at SQLBI.COM)
Alberto Ferrari (SQL Server MVP and BI Consultant at SQLBI.COM)

Reviewers:

Kasper de Jonge, Senior Program Manager, Microsoft

Contents

Introduction

DirectQuery is a technology introduced in Microsoft SQL Server Analysis Services 2012 (SSAS) Tabular models. It transforms the Tabular model into a metadata layer on top of an external database. In fact, by using DirectQuery, the Tabular engine no longer processes data in its own internal database. Instead, the query is transformed into a set of SQL queries that are sent to the underlying relational database. Once the SQL database returns the requested data, the Tabular engine further processes that data. This allows you to create real-time data models, because you no longer need to update the internal data structures. Every query is executed against the relational database, and the data is processed as it becomes available from the SQL query. By using cartridge interfaces, the Tabular engine can connect to various database engines, including Microsoft SQL Server, Oracle, Teradata, and others.

In the 2012/2014 DirectQuery implementation, the Tabular engine generated a single SQL query from any DAX query. This resulted in verbose SQL code that was too complex for most real-world scenarios. Thus, DirectQuery was not widely used in previous SSAS versions. In 2016, Microsoft overhauled the DirectQuery technology. The resulting implementation is much more advanced and, consequently, both faster and more complex to understand.

Before using DirectQuery to turn your SSAS 2016 Tabular model into a real-time model, you first need to understand how DirectQuery works. It's important to understand implementation details as well as the pros and cons of using the technology.

This whitepaper gives you all the relevant information needed to understand and use DirectQuery in your environment. We strongly encourage you to read it from cover to cover before beginning your implementation. DirectQuery comes with some limitations in the modeling and DAX options that will affect how you build the data model itself. DirectQuery requires a different data model than standard Tabular models that import data from an in-memory database.

Note: although the Tabular engine is available in the SQL Server 2016 Standard edition, DirectQuery is an advanced feature available only in the Enterprise edition.

Introduction to Tabular processing

Before discussing the differences between DirectQuery and a regular Tabular model, it's important to understand how Tabular processing and querying works. This paper assumes you are familiar with Tabular models, but gives a brief recap of Tabular processing and querying. For more information on Tabular models, see the resources section at the end of this paper.

A SQL Server Analysis Services (SSAS) Tabular database is a semantic model that usually keeps a copy of the data in an in-memory columnar database, which reads data from data sources, processes them into its internal data structure, and finally answers queries by reading its internal data model. By using DirectQuery, the SSAS Tabular database behaves as a semantic model that translates incoming queries to the data source, without keeping a copy of the data in an in-memory database.

The columnar database used by SSAS is an in-memory engine (VertiPaq). The data sources are typically relational databases, but due to the many different data sources available for SSAS, you can load data into SSAS from virtually any data source, including text files, web services, or Excel workbooks. You can use and mix any of the available data sources because of the intermediate processing phase.

You typically create a SSAS Tabular solution in one of two ways:

- Feed the SSAS solution from the data warehouse that contains all the data relevant to your company, which is already prepared for analysis. In this case, you typically have a single source of data, and this data source is a relational database (can be Microsoft SQL Server or any supported relational database).
- Feed the SSAS solution from multiple data sources (and probably different data types), and use SSAS to integrate data from different databases. In this case, you typically have some relational databases, maybe more than one, and other data sources like text files, Excel files, or other data sources.

In both cases, data is read from the source database and transformed into an in-memory database—highly optimized for querying, and compressed to use less memory. Remember, SSAS engine is the in-memory columnar database that stores and hosts your BI model, and data compression is important. Data is initially saved to disk and loaded when the database is first accessed after a SSAS service restart. After that, all queries are run in RAM, unless you have paging active, which is not recommended for an in-memory engine. Once the data has been processed and stored in memory, you no longer need to connect to the source database(s).

The processing phase comes with the following advantages:

- Data is compressed and stored in a format that makes queries much faster.
- Data can come from different data sources and be transformed into a single format.
- The in-memory engine contains several optimizations for memory access, because all the data is stored in RAM.

The processing phase has the following disadvantages:

- Processing takes time, so real-time queries are not an option. By implementing sophisticated techniques, it's possible to build near real-time models with latency times in the minutes. However, there is no way to make sure that the query executed by SSAS Tabular references the latest changes to the original data source.
- Processing requires a lot of CPU power. During data processing, the server is busy and typically doesn't have the resources to answer queries efficiently.
- Since this is an in-memory database, if the database you are working on does not fit into memory, you'll need to buy more RAM (best option), or optimize the data model's memory usage, which is a complex task.
- Data needs to be moved from the source database to the SSAS storage. When dealing with large amounts of data, just moving the data over the network can take a significant amount of time.

Pros & cons

When using DirectQuery, pros become cons and vice versa. In fact, if you build a model that is enabled to use DirectQuery, your model will not have any in-memory (VertiPaq) storage, and it will not need to process data. Thus, data is always real-time—there is no processing time, no memory limitations, and data does not need to be moved from the source database to the SSAS database. On the other hand, you lose the tremendous speed of the in-memory engine. You will not be able to integrate data coming from different data sources, and the effort of answering queries moves from SSAS to the database engine hosting the information.

There is no golden rule to tell you if DirectQuery is better for your model than regular in-memory storage. You will need to carefully balance advantages and disadvantages and, once you have decided, work through the optimization of your model, which depends on the technology you used.

Example comparing a model designed for in-memory versus DirectQuery

In-memory model (VertiPaq)

During processing, the SSAS engine will execute a `SELECT` statement over the entire table, reading all the rows and performing its own processing steps. This means you need to optimize your data source for a huge single scan of the table. Indexes are useless, and partitioning, if needed, should be aligned with partitions defined in your SSAS solution.

DirectQuery model

If the same data need to be used by DirectQuery, then your table will be accessed at least once for each query and, out of the whole table, only a small subset of it might be needed for the query. In this case, you need to optimize the SQL model to quickly answer the queries generated by DirectQuery. This includes creating the correct indexes on the table, and probably partitioning it to reduce I/O activity during query execution.

How does the model impact decisions?

If you plan to use an in-memory engine, then a columnstore index on a table stored in Microsoft SQL Server is far from ideal. If you plan to use DirectQuery, then the same columnstore index is a much-needed option. As you see, it's important to design your database to fit your use case.

The previous example showed the kind of decisions you'll have to make when implementing DirectQuery into your solution. In the next section, we'll further explore the subtle differences between the DirectQuery and in-memory models.

Introduction to DirectQuery

Now that you've seen how the DirectQuery and in-memory (VertiPaq) models process data, we'll look at how they handle queries.

Every query sent to a Tabular model is executed by two layers of calculation, called *Storage Engine* (SE) and *Formula Engine* (FE). The storage engine is responsible for retrieving data from the database, whereas the formula engine uses the information returned by the storage engine and performs more advanced calculations. For example, if you want to retrieve the top three products by sales amount, SE accesses the source database and computes the list of all products along with the sales amount for each product, whereas FE sorts the resulting data set and retrieves the first three products. Thus, SE reads data from the source database while FE reads data from SE.

Going deeper, Analysis Services parses both DAX and MDX queries, and then transforms them in query plans that are executed by the formula engine. The formula engine is able to execute any function and operation for both languages. In order to retrieve the raw data and perform calculations, the formula engine makes several calls to the storage engine. In SSAS 2016, the storage engine can be a choice between the in-memory analytics engine (VertiPaq) and the external relational database (DirectQuery). You choose which of the two storage engines to use at the data model level. This means that a data model can use only one of the two engines, but not both in the same model.

As you see in Figure 1, the VertiPaq database contains a cached copy of the data that was read from the data source when you last refreshed the data model. In contrast, DirectQuery forwards requests to the external data source when necessary, allowing for real-time queries. The VertiPaq engine accepts requests in internal binary structures (externally described using a human-readable format called xmSQL), whereas DirectQuery cartridges accept requests using the SQL language, in the dialect supported by the cartridge itself.

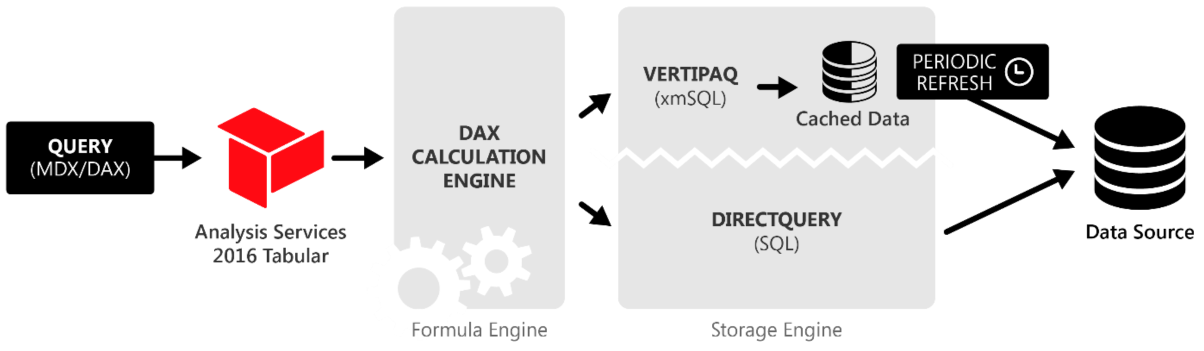


Figure 1 DirectQuery architecture in SQL Server 2016 Analysis Services for tabular models

This is different than how things worked in SSAS 2012/2014. In fact, in previous versions of SSAS, you had the option of building hybrid models, where the client tool had the option of running a query in either standard VertiPaq mode or in DirectQuery mode, as you see in Figure 2.

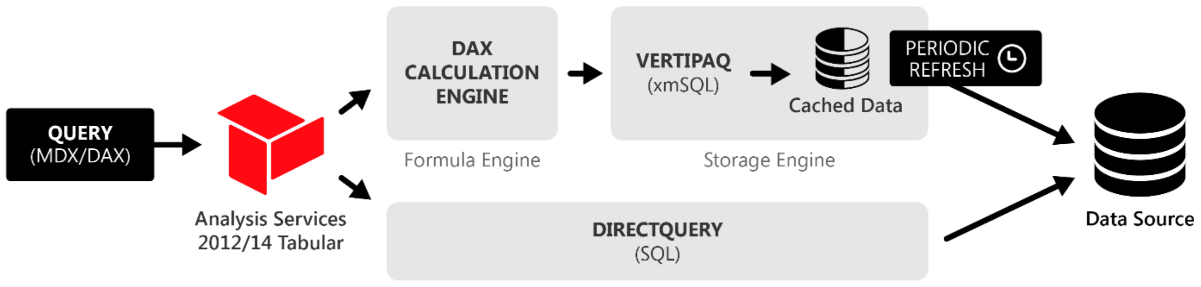


Figure 2 Former DirectQuery architecture in SQL Server Analysis Services 2012/2014 for tabular models

In SSAS 2016, the hybrid mode is no longer available, but this is not a big limitation. As we said earlier, the use of DirectQuery in SSAS 2012/2014 was limited to a few specific scenarios. The rest of this whitepaper will only cover DirectQuery in SSAS 2016.

DirectQuery interacts with relational databases and, in order to use the database at its best, it uses cartridges to support the specific dialect of the server that it uses. At the time of writing, the available cartridges are:

- Microsoft SQL Server (version 2008 or later)
- Microsoft SQL Azure Database
- Microsoft Azure SQL Data Warehouse
- Microsoft Analytics Platform System (APS)
- Oracle (version 9i or later)
- Teradata (V2R6, V12)

For a current list of supported databases and versions, visit:
https://msdn.microsoft.com/en-us/library/gg492165.aspx.

Calculated tables and columns

As explained earlier:

- An in-memory database engine (VertiPaq) can store data.
- DirectQuery is an interface to an existing database. It can only provide data that already exists in the source system.

The difference between VertiPaq and DirectQuery is important when considering calculated tables and columns:

- Calculated tables—DirectQuery does not support calculated tables, mainly because there is no place to store them.

- Calculated columns—You can use calculated columns with DirectQuery, although with some limitations that we'll describe later.

DAX

Some DAX functions have different semantics, because they are converted in correspondent SQL expressions instead of being executed by the in-memory engine. Thus, you might observe inconsistent behavior across platforms when using **Time Intelligence** functions and **Statistical** functions. There are also DAX functions not supported in DirectQuery, and the SSDT designer reports that when you switch a model to DirectQuery.

MDX

MDX has some limitations in DirectQuery that affect only the MDX coding style. You cannot use:

- Relative names
- Session-scope MDX statements
- Tuples with members from different levels in MDX subselect clauses

Other limitations affect the data model design. For example, you cannot reference user-defined hierarchies in an MDX query sent to a model using DirectQuery. This impacts the usability of DirectQuery from Excel, because the feature works without any issue when you use an in-memory storage engine.

Understanding DirectQuery

In this section, we provide a more complete description of the DirectQuery overall architecture, and explain in more detail how the DirectQuery technology works.

Introducing the DirectQuery architecture

As we described in the introduction, DirectQuery is an alternative to an in-memory storage engine (VertiPaq). In SSAS, there is a single formula engine and two different storage engines, as already seen in Figure 1.

Remember, the storage engine is in charge of retrieving data sets from the database, whereas the formula engine performs calculations on top of the data returned by the storage engine. The two engines need to work closely together in order to provide the best performance. In fact, a clear separation between the two would produce sub-optimal query plans. Here is an example.

```
SalesAmt := SUMX ( Sales, Sales[Quantity] * Sales[Unit Price] )
```

The measure references two columns from the database—**Quantity** and **Unit Price**. If the two engines were completely separated, then this is how SSAS would resolve the query:

- The formula engine asks the storage engine for a data set with **Quantity** and **Unit Price**.

- Once the data set is returned, the formula engine iterates on it, performs the calculation (a simple multiplication, in this case), and finally provides the result.

In SQL, this plan would result in a single SQL query like the one shown here:

```
SELECT
    SalesQuantity,
    UnitPrice
FROM
    Sales
```

As you might imagine, for a **Sales** table with hundreds of millions of rows, this operation would require the storage engine to return a large amount of data. The time needed to allocate RAM and transfer data from the database would be significant. In reality, saying that the storage engine can only retrieve data from the database, leaving all computation to the formula engine, is too strong a statement. In fact, the storage engine is able to perform some computation by itself, even if it does not support all DAX functions. Because the storage engine can easily handle simple multiplication, the query above will be resolved as follows:

- The formula engine asks the storage engine for a data set containing the sum of the multiplication of **Quantity** by **Net Price**.
- The storage engine scans the database, then computes the result and returns a single row.
- The formula engine takes the result and packages it into the final result set.

In SQL, a query executed by the database engine would be very different than the previous one:

```
SELECT
    SUM ( SalesQuantity * UnitPrice ) AS Result
FROM
    Sales
```

This latter query scans the tables and performs multiplication at the same time, immediately returning the result to the caller. As such, it is faster and it uses less RAM, making it more efficient than the previous query.

A clear separation between the two engines is not advisable, because coordinating between the two engines produces better results. At the same time, if the storage engine was capable of resolving most (if not all) of the DAX functions, it could become inefficient—it would manage a high level of complexity, considering that both DAX and MDX are powerful languages and the semantics of their functions is rather complex. Thus, SSAS now balances complexity and speed, providing the in-memory engine with enough power to compute common functions, while leaving the most complex computations to the formula engine.

However, with DirectQuery the scenario is much more complex, because DirectQuery is not a single engine, it is a technology that connects the formula engine to many different SQL databases using their own dialects through cartridges. Different databases might have different semantics for the same function, different sets of functionalities, and different performances for the same kind of query. Besides, the SQL language is much more powerful than xmSQL (used by the in-memory model), so DirectQuery typically has the option of pushing more calculations to the SQL Server. As a result, the query plan generated for an in-memory model is different than the query plan generated for a model using DirectQuery, and every cartridge might provide different performances. As a general rule, SQL is powerful and allows the formula engine workload to be lighter in DirectQuery mode. However, each SQL implementation has its own typical behaviors that you need to test before making your modeling decisions.

Using supported data sources

At the time of writing, DirectQuery mode supports the relational databases and providers listed in the following table. For a current list, visit https://msdn.microsoft.com/en-us/library/hh230898.aspx

Data source	**Versions**	**Providers**
Microsoft SQL Server	2008 and later	OLE DB Provider for SQL Server, SQL Server Native Client OLE DB Provider, .NET Framework Data Provider for SQL Client
Microsoft Azure SQL Database	All	OLE DB Provider for SQL Server, SQL Server Native Client OLE DB Provider, .NET Framework Data Provider for SQL Client
Microsoft Azure SQL Data Warehouse	All	.NET Framework Data Provider for SQL Client
Microsoft SQL Analytics Platform System (APS)	All	OLE DB Provider for SQL Server, SQL Server Native Client OLE DB Provider, .NET Framework Data Provider for SQL Client
Oracle relational databases	Oracle 9i and later	Oracle OLE DB Provider
Teradata relational databases	Teradata V2R6 and later	.Net Data Provider for Teradata

Building models for DirectQuery

In this section, we'll identify and analyze additional differences between solutions based on in-memory (VertiPaq) and DirectQuery models.

All tables in a given model must be based on a single source database. You cannot have multiple data sources for the same model in DirectQuery. The reason is that the SQL code generated by the engine to retrieve data will contain joins between tables, filters, and other complex SQL code that cannot work across multiple servers or multiple databases. Moreover, all the tables must be connected to an SQL statement, which cannot be a stored procedure. In T-SQL, you can use table, views, and table-valued functions; the only limitation is on stored procedures.

If you use Microsoft SQL Server, you might use views referencing tables in other databases and/or servers. This is transparent to Analysis Services and resulting performance depends on the SQL Server configuration.

As best practice, always create views in the SQL source to feed the SSAS model, in order to decouple the Tabular model from the physical structure of the database. DirectQuery is not an exception, and views are very useful here too. Remember, the SQL code used to source your tables will be used more than once. Unlike in-memory mode data models that use the SQL code only once, DirectQuery uses the same SQL code multiple times, as each SSAS query retrieves small subsets of data. For best performance, we recommend avoiding complex joins and convoluted SQL code.

As mentioned earlier, a Tabular model in DirectQuery cannot have calculated tables. Moreover, there are several limitations in the DAX expressions that you can use in calculated columns and in expressions for row-level security (RLS). Generally speaking, in calculated columns and row-level security, you can use DAX functions returning a scalar value and operating in a row context, whereas you cannot use aggregation functions, table functions, and iterators. The point is that the DAX expression of a calculated column or of a row-level security filter has to be translated in a corresponding SQL expression, which is not possible (or at least not efficient enough) when the function depends on concepts that are specific to the DAX language (such as filter context and context transition).

Heirarchies are another modeling. User hierarchies are not supported in MDX queries sent to a model in DirectQuery mode. Thus, even if you can create user hierarchies in the Tabular model, these hierarchies are not visible in a client using MDX (such as Excel), whereas they are available in a client using DAX (such as Power BI).

Understanding query limits

Every time DirectQuery sends a query to SQL, it only retrieves a predefined maximum number of rows, which is, by default, 1 million. This is to limit queries that could run too long. Requesting too much memory on Analysis Services can make it difficult to store an intermediate result during a more complex query.

For example, consider the following DAX query:

```
EVALUATE
ROW ( "Rows", COUNTROWS ( Sales ) )
```

It generates a corresponding SQL query that returns only one row:

```
SELECT COUNT_BIG (*) AS [a0]
FROM (
    SELECT  *
    FROM    Sales
) AS t1
```

In the query above, only one row of data has been moved from SQL Server to SSAS. SQL executed the full query, and it returned a small data set, resulting in good performance. However, other DAX queries might transfer a large number of rows to Analysis Services for further evaluation. For example, consider this DAX query:

```
EVALUATE
ROW (
    "Orders",
    COUNTROWS (
        ALL (
            Sales[Order Number],
            Sales[Order Line Number]
        )
    )
)
```

The SQL query that is generated does not execute the `COUNT` operation on SQL Server. Instead, it transfers a combined list of **Order Number** and **Order Line Number** values to Analysis Services, so that the formula engine can count them. However, in order to avoid the transfer of huge amount of data between the two engines, a `TOP` clause limits the number of rows returned by this query to 1 million, as shown in the following code:

```
SELECT TOP ( 1000001 )
    t1.[Order Number],
    t1.[Order Line Number]
FROM (
    SELECT  *
    FROM    Sales
) AS [t1]
GROUP BY
    t1.[Order Number],
    t1.[Order Line Number]
```

Note: The statement returns Order number and Order Line Number, not all the columns from the table. If the result is greater than 1 million rows, the number of rows transferred is exactly one million and one (1,000,000,001), which is a truncated data set. When this happens, SSAS assumes that other rows may not have been transferred, and returns this error:

```
The resultset of a query to external data source has exceeded the maximum allowed size of
'1000000' rows.
```

This default limit of 1 million rows is the same used for models created by Power BI Desktop. This limit is present to prevent huge amount of data being moved between the engines. This is a safety feature, but it can result in queries that cannot be executed. For this reason, you might want to increase this setting on your SSAS instance. To do that, you have to manually edit the msmdsrv.ini configuration file, specifying a different limit for the **MaxIntermediateRowsetSize** setting, which has to be added to the file using the following syntax, because it is not present by default:

```
<ConfigurationSettings>
. . .
<DAX>
  <DQ>
     <MaxIntermediateRowsetSize>1000000
     </MaxIntermediateRowsetSize>
  </DQ>
</DAX>
. . .
```

You can find more details about this and other settings for DAX in the MSDN documentation online at https://msdn.microsoft.com/en-us/library/mt761855.aspx.

Tip! If you have a SSAS Tabular server with a good amount of memory and good bandwidth for connecting to the data source in DirectQuery mode, you probably want to increase this number to a higher value. As a rule of thumb, this setting should be higher than the larger dimension in a star schema model. For example, if you have 4 millions products and 8 million customers, you should increase the **MaxIntermediateRowsetSize** setting to 10 million. In this way, any query aggregating the data at the customer level would continue to work. Using a value that is too high (such as 100 million) could exhaust the memory and/or timeout the query before the limit is reached, so a lower limit helps avoid such a critical condition.

Using DAX in DirectQuery

As mentioned earlier, using DAX has limitations. DirectQuery can provide two different types of support for DAX functions in **measures** and **query expressions**:

- **DAX functions not optimized for DirectQuery**: these functions are not converted in corresponding SQL expressions, so they are executed inside the formula engine. As a consequence, they might require transferring large amounts of data between the source database and the SSAS engine.
- **DAX functions optimized for DirectQuery**: these functions are converted in a corresponding syntax in SQL language, so that their execution is in charge of the specific DirectQuery cartridge. They provide good performance, because they will use the native SQL dialect, avoiding the transfer of large amounts of data between the source database and the formula engine.

Each DAX function optimized for DirectQuery belongs to one of two groups:

Group 1: DAX functions that are also available in calculated columns and in filter expressions of row-level security. These functions are optimized for DirectQuery and are supported in all DAX formulas.

They include:

ABS, ACOS, ACOT, AND, ASIN, ATAN, BLANK, CEILING, CONCATENATE, COS, COT, CURRENCY, DATE, DATEDIFF, DATEVALUE, DAY, DEGREES, DIVIDE, EDATE, EOMONTH, EXACT, EXP, FALSE, FIND, HOUR, IF, INT, ISBLANK, ISO.CEILING, KEEPFILTERS, LEFT, LEN, LN, LOG, LOG10, LOWER, MAX, MID, MIN, MINUTE, MOD, MONTH, MROUND, NOT, NOW, OR, PI, POWER, QUOTIENT, RADIANS, RAND, RELATED, REPT, RIGHT, ROUND, ROUNDDOWN, ROUNDUP, SEARCH, SECOND, SIGN, SIN, SQRT, SQRTPI, SUBSTITUTE, SWITCH, TAN, TIME, TIMEVALUE, TODAY, TRIM, TRUE, TRUNC, UNICODE, UPPER, USERNAME, USERELATIONSHIP, VALUE, WEEKDAY, WEEKNUM, YEAR.

Group 2: DAX functions that cannot be used in calculated columns or filter expressions, but that can be used in measures and queries. This group includes DAX functions that are optimized for DirectQuery and supported only in measures and query formulas, but cannot be used in calculated columns and row-level security filters:

They include:

ALL, ALLEXCEPT, ALLNOBLANKROW, ALLSELECTED, AVERAGE, AVERAGEA, AVERAGEX, CALCULATE, CALCULATETABLE, COUNT, COUNTA, COUNTAX, COUNTROWS, COUNTX, DISTINCT, DISTINCTCOUNT, FILTER, FILTERS, HASONEFILTER, HASONEVALUE, ISCROSSFILTERED, ISFILTERED, MAXA, MAXX, MIN, MINA, MINX, RELATEDTABLE, STDEV.P, STDEV.S, STDEVX.P, STDEVX.S, SUM, SUMX, VALUES, VAR.P, VAR.S, VARX.P, VARX.S.

For a current list of supported DAX functions, visit:
https://msdn.microsoft.com/en-us/library/mt723603.aspx#Anchor_0.

All other DAX functions not included in these two lists are available for DirectQuery measure and query formulas only, but they are not optimized. As a consequence, the calculation could be implemented in the formula engine on Analysis Services, which will retrieve the required granularity from the source database to perform the calculation. Aside from slower performance, executing the query could require materializing a large SQL query result in the Analysis Services memory. For this same reason, if you have complex calculations over large tables, be sure to carefully review the **MaxIntermediateRowsetSize** setting described earlier.

Semantic differences in DAX

Sometimes a DAX expression will produce different results in DirectQuery versus in-memory models. This is caused by differences in semantics between DAX and SQL for:

- Comparisons (strings and numbers, text with Boolean, and nulls)
- Casts (string to Boolean, string to date/time, and number to string)
- Math functions and arithmetic operations (order of addition, use of `POWER` function, numerical overflow, `LOG` functions with blanks, and division by zero)
- Numeric and date-time ranges
- Currency

- `Text` functions

See Appendix A for more detail.

Division by 0 and division by Blank

In DirectQuery mode, division by zero (0) or division by BLANK will always result in an error. SQL Server does not support the notion of infinity, and because the natural result of any division by 0 is infinity, the result is an error. However, SQL Server supports division by nulls, and the result equals to null.

Rather than returning different results for these operations, in DirectQuery mode both types of operations (division by zero and division by null) return an error.

In Excel and in PowerPivot models, division by zero also returns an error. However, division by BLANK returns a BLANK.

For example, the following expressions are valid for in-memory models, but will fail in DirectQuery mode:

```
1/BLANK
1/0
0.0/BLANK
0/0
```

The expression BLANK/BLANK is a special case that returns BLANK in both in-memory and DirectQuery modes.

Statistical functions over a table with a single row

Statistical functions on a table with one row return different results. Aggregation functions over empty tables also behave differently in in-memory models than they do in DirectQuery mode. If the table that is used as an argument contains a single row, in DirectQuery mode, statistical functions such as `STDEV` and `VARx` return *null*.

In an in-memory model a formula that uses `STDEV` or `VARx` over a table with a single row returns a division by zero error.

Using DAX measures

DAX measures are translated either into SQL code or into SQL queries retrieving raw data plus a formula engine query plan, depending on the kind of function you use in the measure itself.

For example, the following query defines the **Sales[Amt] measure**, and retrieves the sales amount for each color.

```
DEFINE
    MEASURE Sales[Amt] =
        SUMX ( Sales, Sales[Quantity] * Sales[Unit Price] )
EVALUATE
SUMMARIZECOLUMNS ( Product[Color], "Amt", [Amt] )
```

This is resolved by the following single SQL query:

```
SELECT TOP (1000001)
    t0.ProductColor,
    SUM ( t0.PriceMultipliedByQuantity ) AS PriceMultipliedByQuantity
FROM
    (SELECT
        Sales.[Quantity] AS Quantity,
        Sales.[Unit Price] AS [Unit Price],
        Product.[Color] AS ProductColor,
        (Sales.[Quantity] * Sales.[Unit Price]) AS PriceMultipliedByQuantity
     FROM
        Sales
        LEFT OUTER JOIN Product
            ON Sales.[ProductKey] = Product.[ProductKey]
    ) AS [t0]
GROUP BY
    t0.ProductColor
```

As you can see, the entire calculation has been pushed down to SQL Server. The data that is moved between SSAS and SQL is only the actual result, whereas the calculation is pushed to the lowest level down the datasource.

However, from this code alone, you cannot deduct that no materialization is happening. In fact, DirectQuery uses SQL to ask the relational database to produce the multiplication result. Inside the relational engine, SQL might still make some materialization. When possible, DirectQuery avoids transferring large amounts of data between SQL and SSAS by asking to perform the calculation in the most efficient way. Then, it's up to the relational database to compute the value. This is important, because it clearly shows that system performance mainly depends on how well the relational database is optimized. DirectQuery has no way to know how the model in SQL is structured. It might be in a heap table or in a clustered columnstore index in Microsoft SQL Server, and it might have the right indexes in place or not. DirectQuery only builds a query. The database administrator is responsible for guaranteeing that the queries generated by DirectQuery will be executed at top speed.

In measures, there is an important difference between optimized and non-optimized function usage. In fact, if you change the previous measure and, instead of `SUMX`, use a non-optimized function like `MEDIANX`, then the scenario changes significantly. For example:

```
DEFINE
    MEASURE Sales[Amt] =
        MEDIANX ( Sales, Sales[Quantity] * Sales[Unit Price] )
EVALUATE
SUMMARIZECOLUMNS ( Product[Color], "Amt", [Amt] )
```

In the third line, the measure uses `MEDIANX`, which is not optimized for DirectQuery. As a result, the engine needs to retrieve, from the database, the values for the **Quantity** and **Unit Price** columns for all rows of the **Sales** table, as shown in the following SQL query:

```
SELECT TOP (1000001)
    Sales.[Quantity] AS Quantity,
    Sales.[Unit Price] AS [Unit Price],
    Product.[Color] AS ProductColor
FROM
    Sales
    LEFT OUTER JOIN Product
        ON Sales.[ProductKey] = Product.[ProductKey]
```

Not only would this result in poor performance, but it also produces an error, because we are querying the fact table in our test database that contains 12 million rows. As explained in the *Understanding Query Limits* section above, DirectQuery has a setting that prevents retrieving more than 1 million rows. Even though the **MaxIntermediateRowsetSize** setting is configurable, you probably don't want to query several million rows every time users evaluate a query or a pivot table. This setting helps avoid long-running queries and improves user experience.

When starting a project, it can be challenging to predict which functions you'll need and what level of materialization is optimal for queries. Do your best to choose functions carefully, and avoid materializing large data sets. It's best to have strong knowledge of DAX before adopting DirectQuery, so that you can evaluate the impact of calculations while in DirectQuery mode. You'll need to rely on your own experience to determine if you need non-optimized functions.

Using DAX calculated columns

In-memory mode (VertiPaq) and DirectQuery data models handle calculated columns in substantially different ways. In-memory mode computes calculated columns at processing time and stores results in the database. DirectQuery, however, is a query layer and not a database, so calculated columns can't be computed during process time (because there is no processing time) and the results can't be stored (because there is no storage space).

When using DirectQuery, calculated columns need to be computed every time you use them. For this reason, you can use only the first subset of optimized functions. Remember, some optimized functions can be used everywhere, and others can be used only in measures and queries. In other words, in DirectQuery calculated columns, you can use only optimized functions that can be computed for every query and that don't need to be stored in the model.

For example, a calculated column in the **Product** table containing a simple SUM like the one shown here, cannot be implemented in DirectQuery:

```
Product[TotalQuantity] = CALCULATE ( SUM ( Sales[Quantity] ) )
```

In fact, the calculated column computes the quantity sold for the given product, but, for efficiency, the results are stored in the **Product** table. Trying to compute this value every time that it's needed will negatively affect the query performance, so a Tabular model in DirectQuery mode does not allow you to define a calculated column in this way.

On the other hand, simple calculations that depend only on the current row can be easily implemented. Thus, a calculated column like the following one works fine:

```
Sales[LineTotal] = Sales[Quantity] * Sales[Unit Price]
```

Nevertheless, keep in mind that the calculation will happen every time you run a query. For example:

```
EVALUATE
SUMMARIZECOLUMNS (
    Product[Color],
    "Amt", CALCULATE ( SUM ( Sales[LineTotal] ) )
)
```

Since the query aggregates a calculated column **(LineTotal)**, it results in the following SQL query being sent to SQL Server:

```
SELECT TOP ( 1000001 )
    Product.[Color],
    SUM ( [LineTotal] ) AS [a0]
FROM
    (SELECT
        Sales.[ProductKey] AS [ProductKey],
        Sales.[Quantity] * Sales.[Unit Price] AS [LineTotal]
     FROM
        Sales
        LEFT OUTER JOIN Product
            ON Product.[ProductKey] = Sales.[ProductKey]
    )
GROUP BY
     Product.[Color]
```

As you can see, every row is calculated every time you query this column in the model. Simple calculations are not an issue. However, for more complex calculations, consider materializing the results in the data source, which will avoid the complexity of calculating every query.

Note that calculated columns in aggregations have the side effect of being computed again and again, but, depending on the size of the database, performance could be good enough. However, if you plan to use calculated columns as filters, then the scenario becomes more complex. For example, here is a simple variation of the query:

```
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS (
        Product[Color],
        "Amt", CALCULATE ( SUM ( Sales[LineTotal] ) )
    ),
    Sales[LineTotal] >= 100
)
```

This time, the query not only aggregates **LineTotal** (as we did before), but also applies a filter to **LineTotal** requiring it to be greater than or equal to 100. Since **LineTotal** is a calculated column, it will be computed at query time. This is the resulting SQL query:

```
SELECT TOP (1000001)
    Product.[Color],
    SUM ( [LineTotal] ) AS [a0]
FROM
    ( ( SELECT
           Sales.[ProductKey] AS [ProductKey],
           ( Sales.[Quantity] * Sales.[Unit Price] ) AS [LineTotal]
        FROM
           Sales
     ) AS A
     LEFT OUTER JOIN Product
        ON (Sales.[ProductKey] = Product.[ProductKey])
    )
WHERE
    ( A.[LineTotal] >= CAST ( N'100' AS MONEY ) )
GROUP BY
    Product.[Color];
```

If you look at the code, you will notice that the filter on **LineTotal** will be applied after a full scan of the fact table. In other words, the only way SQL can resolve the condition is to scan the whole table, compute the value of **LineTotal**, and then remove the rows that do not satisfy the condition. This is not intrinsically bad, but it shows that filtering with a calculated column happens only after a full scan, instead of the filter being applied before the data is retrieved. In other words, the filter is unlikely to be optimized, and no matter how many rows it will retrieve, it probably needs to perform a full scan.

The same scenario with an in-memory mode data model would result in a more optimized query plan, because the value of **LineTotal** would be known in advance and stored in the model. Thus, a filter in the query reduces the time spent in scanning the table.

Of course, if you materialize **LineTotal** in the source database and build appropriate indexes, then the query plan would be optimized in SQL as well, but at the cost of additional storage space for the calculated column value.

When building calculated columns in a DirectQuery model, you need to understand the implications of a calculated column and how the engine is going to implement their usage. Failing to do so might result in bad performance.

Using MDX in DirectQuery

In a Tabular model, you must use DAX to define measures and calculated columns, but the queries can be written using both DAX and MDX. In this way, a Tabular model is compatible with any existing MDX client, such as the PivotTable in Excel, and any DAX client, like Power BI. In DirectQuery mode, there are limitations to the MDX features available. As a rule of thumb, the MDX generated by Excel works in DirectQuery, but these limitations might affect other client products or MDX queries created manually in reports or other tools. The limitations are:

- You cannot use relative object names. All object names must be fully qualified.
- No session-scope MDX statements (named sets, calculated members, calculated cells, visual totals, default members, and so forth) are allowed, but you can use query-scope constructs, such as the `WITH` clause.
- You cannot use tuples with members from different levels in MDX subselect clauses.
- User-defined hierarchies cannot be used.

For a current list of limitations, visit:
https://msdn.microsoft.com/en-us/library/hh230898.aspx#Anchor_1.

Using row-level security

When you have a Tabular model in DirectQuery mode, you can define the security in two places:

- You can use the security roles defined in Analysis Services, just as you do in models using in-memory mode.
- You can define security on the relational data source by instructing Analysis Services to impersonate the current user when it sends the SQL queries to the data source.

Usually, you choose either one technique or the other, but nothing prevents you from combining both, even if it's usually not necessary.

If you want to rely on the standard role-based security provided by Analysis Services, then you have to be aware that all SQL queries will include the necessary predicates and joins to retrieve only the required data. This can result in complex (and potentially slow) queries, unless you pay close attention to the details of how security is defined. Moreover, when you use DirectQuery, there are restrictions in the DAX expressions used in the role filters that are the identical restrictions applied to calculated columns.

If you already have row-level security implemented in the relational database and supporting Windows integrated security, you can configure Analysis Services to impersonate the current user. Doing this, SSAS will execute SQL queries impersonating the user who is connected to SSAS, so that the relational database can recognize the person and apply the necessary security configuration. For example, you can define the row-level security (RLS) in Microsoft SQL Server 2016, and rely on that by impersonating the current user in Analysis Services.

To impersonate the current user while SSAS and the relational database are running on different servers, then you will also need to configure Kerberos delegation to allow the token (of the user impersonated by SSAS) to flow to the relational database. This eliminates the well know "double-hop" problem of Windows security tokens.

Using DirectQuery, real-time, and different client tools

As mentioned before, DirectQuery is a querying system that cannot store information. By nature, DirectQuery is a real-time system, meaning that it queries the data at the moment the query is executed. DirectQuery, by itself, does not perform any kind of caching between different queries.

If you run the same query at different points in time, you can retrieve different results. It doesn't matter if one hour or a tenth of a second separates the two points in time—the results can be different, because, in the meantime, data can change in a real-time system.

At the time of publication, the two main SSAS clients are Excel and Power BI. These two clients use different patterns to query the model, which may return different (and potentially confusing) results depending on which client you are using.

Power BI

Power BI generates DAX queries and has its own caching system. This avoids repeatedly sending the same query to the server whenever the user interacts with the visual components. In fact, when creating reports, users will likely try different visuals. To avoid querying the SSAS server every time a user clicks on a visualization, Power BI caches the results of visuals on the client, and reuses them for some time. This works well for in-memory mode data models, because data remains static until it is explicitly refreshed with a process. However, in a real-time system, data changes in a continuous way.

For example, imagine a user wants to create a visual showing total sales, and the initial query returns 1,000 sales. A few seconds later, the user builds a new matrix with sales divided by color, but now the grand total shows 1,001. This happens because, in the time between the first and second queries, a new sale was added to the database. Until the report is refreshed, or the cache expires, the visuals will not be updated. This behavior might be confusing, because different visuals can show different totals, but it is expected behavior for standard caching systems.

You have two options:

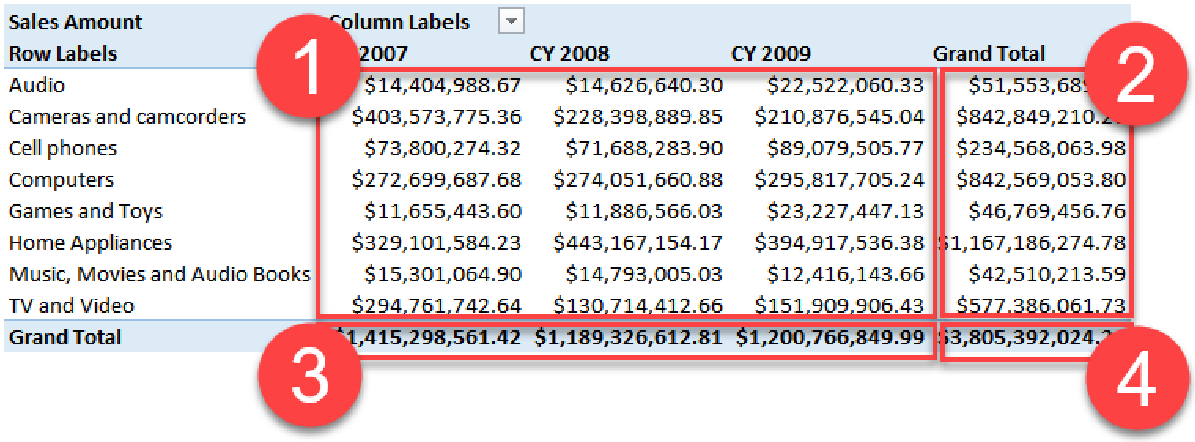
- Avoid the cache, which decreases performance, but increases accuracy.
- Use the cache, which is faster due to decreased latency, but may return stale data.
  In-memory mode data models only return stale data if the server processes the database between two queries. This rarely occurs. Note: you can clear the Power BI cache by clicking the Refresh button.

Excel

Excel uses MDX to query the model, and does not have any client-side caching. Every time you update a PivotTable, the entire query is executed. No cached data is used and the results are highly accurate, but the source data can change between queries, which can result in confusing visualizations.

For Example: Excel PivotTables have both detailed and aggregated information. Here, we have highlighted four different areas in the PivotTable:

1. Raw sales data for each category and year.
2. Grand total at the row level, which is the total for the category regardless of the year.
3. Grand total at the column level, which is the total for the year, regardless of the category.
4. Grand total of the PivotTable, which includes any year and category.



Sales Amount	olumn Labels			
Row Labels	2007	CY 2008	CY 2009	Grand Total
Audio	$14,404,988.67	$14,626,640.30	$22,522,060.33	$51,553,68
Cameras and camcorders	$403,573,775.36	$228,398,889.85	$210,876,545.04	$842,849,210.
Cell phones	$73,800,274.32	$71,688,283.90	$89,079,505.77	$234,568,063.98
Computers	$272,699,687.68	$274,051,660.88	$295,817,705.24	$842,569,053.80
Games and Toys	$11,655,443.60	$11,886,566.03	$23,227,447.13	$46,769,456.76
Home Appliances	$329,101,584.23	$443,167,154.17	$394,917,536.38	1,167,186,274.78
Music, Movies and Audio Books	$15,301,064.90	$14,793,005.03	$12,416,143.66	$42,510,213.59
TV and Video	$294,761,742.64	$130,714,412.66	$151,909,906.43	$577,386,061.73
Grand Total	**1,415,298,561.42**	**$1,189,326,612.81**	**$1,200,766,849.99**	**3,805,392,024.**

Figure 3 Sample PivotTable with the different queries highlighted

There is no efficient way to execute a single SQL query returning all of this information in a single step. In fact, to fill the PivotTable, the Excel MDX code is transformed into a set of queries that are sent to the SQL server, each of which retrieves one of the results. In total, DirectQuery will send at least four SQL queries to populate this PivotTable. Obviously, in a more complex report, the number of queries is much higher.

If the first query takes one second, then the second query is executed one second later, which is a different point in time. The same happens for the third and the fourth queries. As a real-time system with no cache, DirectQuery will return – for each query – the data available at that point in time. The data might change between queries resulting in a PivotTable where the total is not the sum of the individual lines.

How frequently this will occur is unpredictable and depends on:

- The database — How frequently is it updated?
- The data — How recent is the data?
- The queries — How fast does each individual SQL query run?
- Note: if you choose to use DAX, a single DAX query might also result in many different SQL queries executed at different points in time. For example, look at this simple query:

```
EVALUATE
SUMMARIZECOLUMNS (
    ROLLUPADDISSUBTOTAL ( 'Product'[Color], "IsSubtotal" ),
    "Amt", SUMX ( Sales, Sales[Quantity] * Sales[Net Price] )
)
```

Using the same sample PivotTable in Figure 3, because the query uses the `ROLLUPADDISSUBTOTAL` function, you obtain both the individual rows and the grand total. The grand total is generated through a different SQL query executed at a different point in time, so the results may not match.

Note: sometimes, simple additive calculations can result in a single SQL query. Nevertheless, as a general rule, you should assume that different granularities in the result generate different queries to the source database.

Do you need real-time systems?

Any discussion about DirectQuery architecture is incomplete without understanding real-time and analytical systems. As already discussed, real-time systems (like DirectQuery) suffer from several issues:

- There are limitations in the modeling capabilities.
- There is no processing and consolidation point in time.
- Cache usage is limited and, when present, can cause issues.
- Report results are volatile and might change with every refresh.
- A single report might return inconsistent numbers for the individual lines and the totals, or for different visuals in the same page.

These limitations come with two big advantages:

- There is no processing, which eliminates the need to replicate data in different databases.
- Data is always current, which eliminates the need to manage the complexity of near-real-time systems.

Every environment is different. Questions to consider:

- Do I need real-time results?
- Do I need real-time results for the entire data model?
- Do I need a single—real-time—data model?

In most situations, real-time requirements only affect the most recent few days, and involve only simple calculations over this limited time period. For example, you might want to build a dashboard showing the number of claims resolved in the last week, day, and hour. However, it's unlikely that you'll need a single data model containing both claims data for the last 10 years as well data for current claims, including those made today. This is especially true in cases where the real-time requirements for the current claims' key performance indicators (KPI) impose restrictions to the entire data model.

We recommend separating the real-time requirements (last few days) from the analytical requirements (several years) by building two models:

- **A real-time model for recent data (the last few days)** — This database will be smaller, and will have all the limitations of real-time models, but also will have much simpler calculations. This model is based on the DirectQuery technology.
- **An in-memory mode model for historical data (all data from the last 10 years)** — The standard in-memory mode model has full SSAS power with few limitations on complex calculations.

The in-memory mode, historical model will likely need complex DAX code, time intelligence calculations, complex calculated columns, customer segmentation, and the usual complex formulas seen in Business Intelligence solutions. The real-time model tends to be less complex, because it usually consists of simple counts, percentages, and basic information that spotlights what is happening right now.

Managing multiple data models can seem like more work, but choosing the right model(s) up front can save you a lot of time later. We recommend carefully evaluating the pros and cons of differing SSAS technologies against your requirements. Don't be shy to use more than one model if it's the right thing for your implementation.

Creating a DirectQuery model

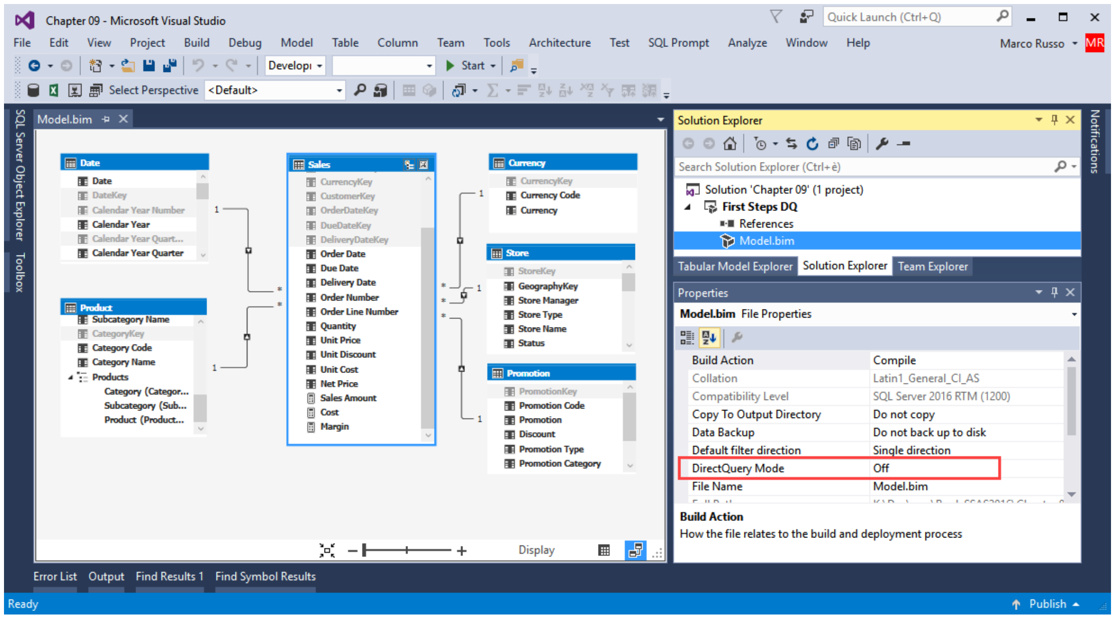
Now that we have a clear picture of DirectQuery—its overall architecture, main limitations, and the different ways queries are resolved—it's time to introduce more practical topics and see how to create a DirectQuery model. This will not be a step-by-step guide on how to build an SSAS solution. Instead, we will show you how to configure the DirectQuery settings of an existing, simple solution that you can download here: <https://go.microsoft.com/fwlink/?linkid=841809>.

Enabling DirectQuery is a single setting at the data model level, which affects the entire project. You can enable DirectQuery in two ways:

- **During development**—Activate the **DirectQuery Mode** setting in the model properties using SQL Server Data Tools (SSDT).
- **After deployment**—Set the model's **Default Mode** property to DirectQuery using SQL Server Management Studio (SSMS), or apply an equivalent change using PowerShell, TMSL, or XMLA.

Enabling DirectQuery during development

When you create a new data model using SSDT, by default the **DirectQuery Mode** setting is set to *Off* (see Figure 3). This means that data in all tables is imported into an in-memory mode data model. A data preview is shown in Grid view.


Chapter 09 - Microsoft Visual Studio
File Edit View Project Build Debug Model Table Column Team Tools Architecture Test SQL Prompt Analyze Window Help
Marco Russo
Select Perspective <Default>
Model.bim
Date
Sales
Currency
Store
Product
Promotion
Solution Explorer
Solution 'Chapter 09' (1 project)
First Steps DQ
References
Model.bim
Tabular Model Explorer Solution Explorer Team Explorer
Properties
Model.bim File Properties
Error List Output Find Results 1 Find Symbol Results
Ready
Publish

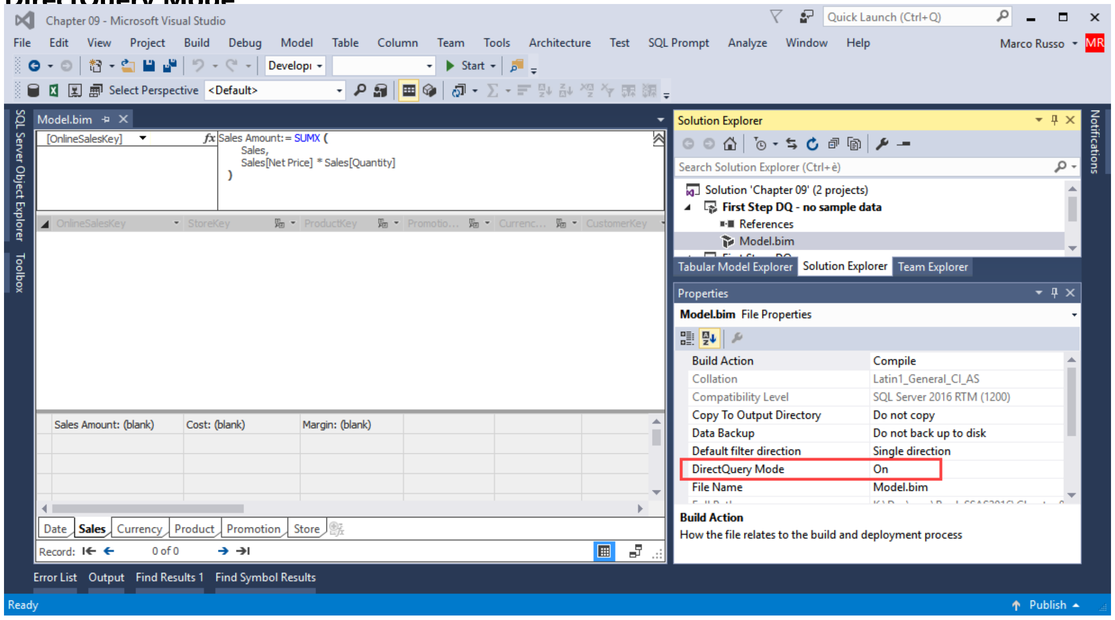
Build Action	Compile
Collation	Latin1_General_CI_AS
Compatibility Level	SQL Server 2016 RTM (1200)
Copy To Output Directory	Do not copy
Data Backup	Do not back up to disk
Default filter direction	Single direction
DirectQuery Mode	Off
File Name	Model.bim

Build Action
How the file relates to the build and deployment process

Figure 3 DirectQuery Mode property is available at the model level in Visual Studio

To switch the data model to DirectQuery, set the **DirectQuery Mode** property to *On* (see Figure 4). In the Grid view, the tables do not show any data. In the data grid preview, all measures are blank, because the model no longer has any data in memory.

DirectQuery Mode



```
Sales Amount := SUMX (
    Sales,
    Sales[Net Price] * Sales[Quantity]
)
```

Sales Amount: (blank) | Cost: (blank) | Margin: (blank)

Model.bim File Properties

Build Action	Compile
Collation	Latin1_General_CI_AS
Compatibility Level	SQL Server 2016 RTM (1200)
Copy To Output Directory	Do not copy
Data Backup	Do not back up to disk
Default filter direction	Single direction
DirectQuery Mode	On
File Name	Model.bim

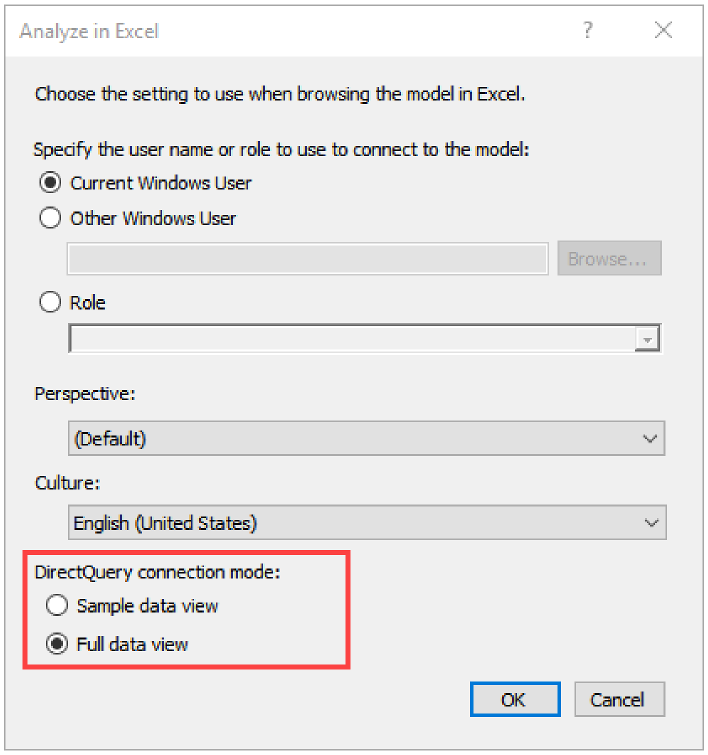
Build Action
How the file relates to the build and deployment process

Figure 4 Switching property to *On*; data preview no longer visible in Grid view

We strongly recommend turning **DirectQuery Mode** to *On* before importing the table into memory. If **DirectQuery Mode** is *Off* when the tables are imported, all that data will be lost as soon as you switch **DirectQuery Mode** to *On*.

To browse the data through the workspace database:

1. In the Excel menu, select *Model/Analyze*
2. In the *Analyze in Excel* dialog box, find the *DirectQuery connection mode* section, and choose *Full Data View* (see Figure 5). By default, this is set to *Sample Data View*, but you do not yet have defined sample data.


Analyze in Excel

Choose the setting to use when browsing the model in Excel.

Specify the user name or role to use to connect to the model:

- [✓ Current Windows User]
- Other Windows User
- Role

Perspective: (Default)

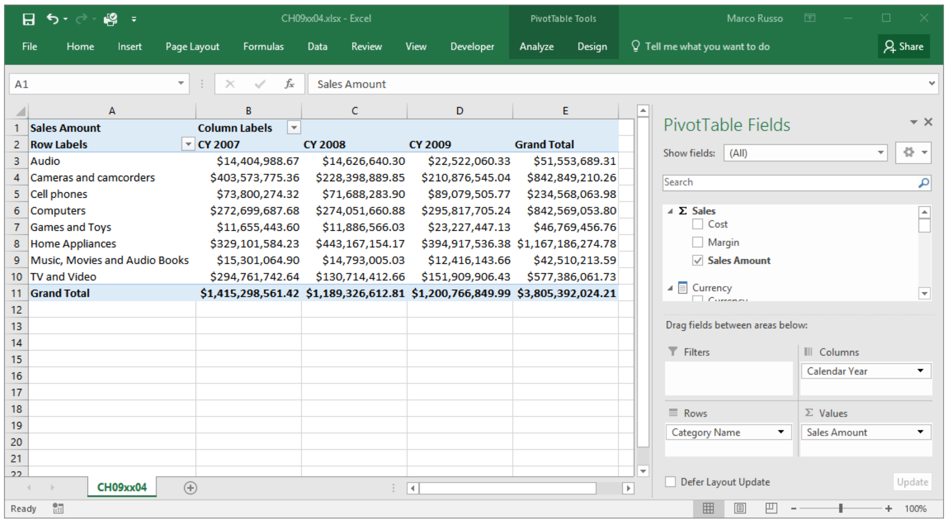
Culture: English (United States)

DirectQuery connection mode:

- Sample data view
- [✓ Full data view]

Figure 5 Setting DirectQuery connection mode

Once in Excel, you can browse the data model with a PivotTable, just as you do with an in-memory mode data model. You'll notice that performance is poor, because the solution has not been optimized for DirectQuery. For example, the PivotTable shown in Figure 6 might require several seconds to refresh (around 30 seconds is normal, but you might wait more than one minute, depending on your SQL Server configuration).



Sales Amount

Sales Amount	Column Labels			
Row Labels	CY 2007	CY 2008	CY 2009	Grand Total
Audio	$14,404,988.67	$14,626,640.30	$22,522,060.33	$51,553,689.31
Cameras and camcorders	$403,573,775.36	$228,398,889.85	$210,876,545.04	$842,849,210.26
Cell phones	$73,800,274.32	$71,688,283.90	$89,079,505.77	$234,568,063.98
Computers	$272,699,687.68	$274,051,660.88	$295,817,705.24	$842,569,053.80
Games and Toys	$11,655,443.60	$11,886,566.03	$23,227,447.13	$46,769,456.76
Home Appliances	$329,101,584.23	$443,167,154.17	$394,917,536.38	$1,167,186,274.78
Music, Movies and Audio Books	$15,301,064.90	$14,793,005.03	$12,416,143.66	$42,510,213.59
TV and Video	$294,761,742.64	$130,714,412.66	$151,909,906.43	$577,386,061.73
Grand Total	**$1,415,298,561.42**	**$1,189,326,612.81**	**$1,200,766,849.99**	**$3,805,392,024.21**

Sales
- Cost
- Margin
- [✓ Sales Amount]

Currency

Filters:

Columns: Calendar Year

Rows: Category Name

Values: Sales Amount

Figure 6 PivotTable using Full Data View of DirectQuery connection mode

This first example is behaving as expected. It is slow, because the Analysis Services engine is requiring SQL Server to handle most of the calculations. Optimizing the calculation becomes a matter of optimizing the SQL Server database for the typical workload produced by DirectQuery, which is quite different from the type of queries sent by an in-memory mode data model to process in an in-memory database. A columnstore index can be useful in this situation, and we'll provide more details on this topic later in this whitepaper.

Also, there are no user hierarchies in DirectQuery. For example, the original Tabular model used in this example has a *Products* hierarchy in the **Product** table. The hierarchy is not deleted when you turn DirectQuery on, yet it's not available when you browse the data using Excel, because of the limitations described previously. However, *Hierarchies* are available in Power BI.

Using DirectQuery in the development environment could be hard if you don't have data, and if all the queries executed to test the model are particularly slow. (For example, you might have a data source for development that doesn't perform as well as the data in the production environment.) For this reason, you might define additional partitions in the data model to use as sample data. If you provide sample data partitions, Analysis Services will use the partitions loaded in memory with sample data, and will show only this content to the user. It's up to you to define what content to use as sample data in every table. By default, no tables in a Tabular model have sample data. Thus, if you try to connect to a model using *Model/Analyze* in the Excel menu again, and you choose the *Sample Data View* for *DirectQuery connection mode* instead of *Full Data View* in the *Analyze In Excel* dialog box, you will obtain a PivotTable with only a list of measures, tables, and columns, but without any data. In the following section, you learn how to add sample data for DirectQuery to your tables.

Creating sample data for DirectQuery

For each table, you can define one or more partitions with sample data. Partitions can help reduce processing time. The table data is the sum of all its partitions. However, a model running in DirectQuery mode only uses one partition to identify the corresponding table object within the relational database that is being used as a data source. This table is marked *DirectQuery*, and only one partition can be marked as *DirectQuery*. In a model enabled for DirectQuery, all the other partitions of a table contain only sample data.

Steps to add a partition with sample data:

1. Open the *Partition Manager* window selecting the *Table/Partitions* menu item.
2. Select the partition you want to copy, and then click the *Copy* button.
3. In the *Partition Name* field, rename the suffix from *Copy* to *Sample* (*Sample* is a suggestion; you can use name.)
4. Optional: For large tables, apply a filter to the SQL statement by clicking on the SQL button and applying a `WHERE` condition. This loads only a subset of rows into the Sample partition, which reduces the memory requirements and data processing time.

In our example, we copy data from all the tables into a new Sample partition following the first three steps above, and only for the **Sales** table we add a `WHERE` condition to filter only the first day of each month (see Figure 7).

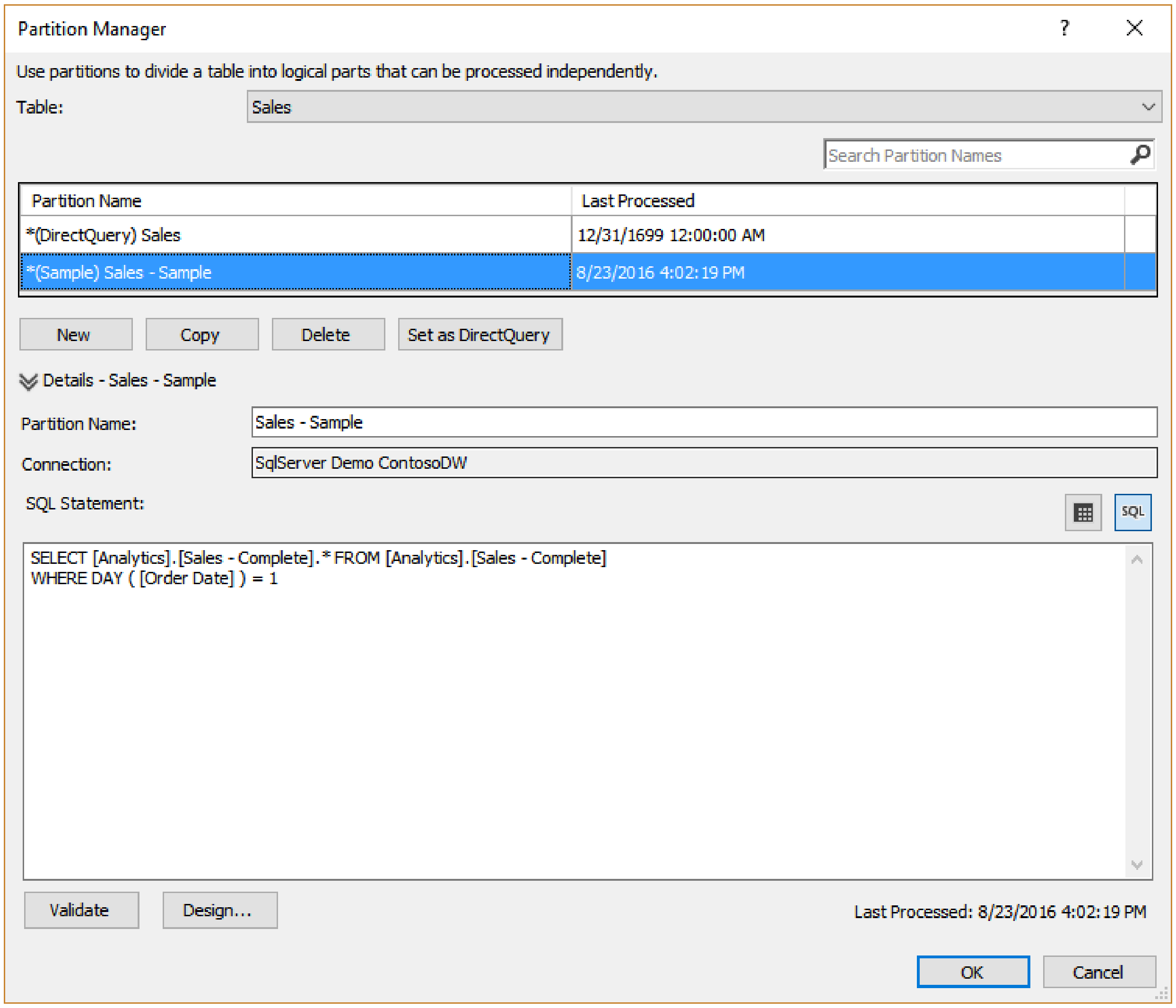


Table: Sales

Partition Name	Last Processed
*(DirectQuery) Sales	12/31/1699 12:00:00 AM
*(Sample) Sales - Sample	8/23/2016 4:02:19 PM

Details - Sales - Sample

Partition Name: Sales - Sample

Connection: SqlServer Demo ContosoDW

SQL Statement:

```
SELECT [Analytics].[Sales - Complete].* FROM [Analytics].[Sales - Complete]
WHERE DAY ( [Order Date] ) = 1
```

Last Processed: 8/23/2016 4:02:19 PM

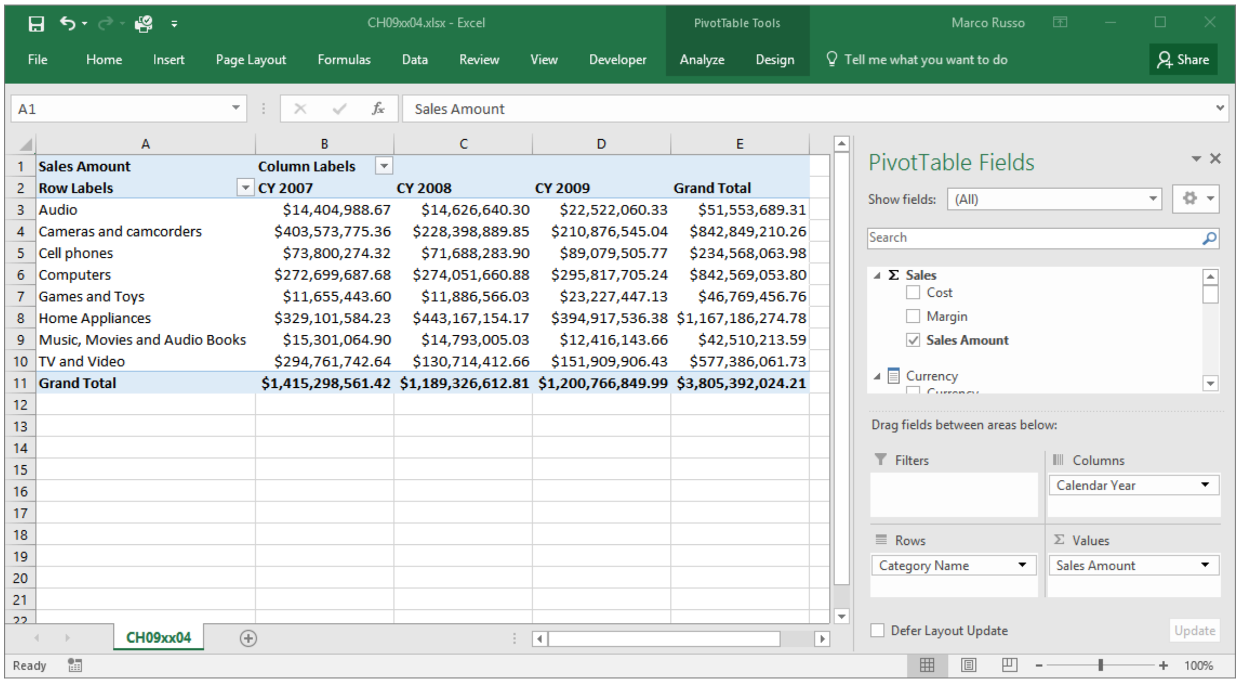
Figure 7 Partition Manager dialog box with one sample partition in a model enabled for DirectQuery

If you want to disable DirectQuery Mode, you must first remove all sample partitions. An in-memory mode data model cannot contain sample data.

Steps to populate sample partitions:

1. Process the workspace database tables by clicking on *Model/Process/Process* All in the menu.
2. Once the tables are processed, go to the Excel menu and select *Model/Analyze*.
3. In the *Analyze in Excel* dialog box, go to the *DirectQuery connection mode* section, and again choose *Full Data View*.
4. Browse data using a PivotTable.

The response times should be faster than before. Also, the numbers are smaller than those seen in *Full Data View* (see Figure 8), because you are querying only the partitions with sample data, and you are not using DirectQuery in this PivotTable.



Sales Amount	Column Labels			
Row Labels	CY 2007	CY 2008	CY 2009	Grand Total
Audio	$14,404,988.67	$14,626,640.30	$22,522,060.33	$51,553,689.31
Cameras and camcorders	$403,573,775.36	$228,398,889.85	$210,876,545.04	$842,849,210.26
Cell phones	$73,800,274.32	$71,688,283.90	$89,079,505.77	$234,568,063.98
Computers	$272,699,687.68	$274,051,660.88	$295,817,705.24	$842,569,053.80
Games and Toys	$11,655,443.60	$11,886,566.03	$23,227,447.13	$46,769,456.76
Home Appliances	$329,101,584.23	$443,167,154.17	$394,917,536.38	$1,167,186,274.78
Music, Movies and Audio Books	$15,301,064.90	$14,793,005.03	$12,416,143.66	$42,510,213.59
TV and Video	$294,761,742.64	$130,714,412.66	$151,909,906.43	$577,386,061.73
Grand Total	**$1,415,298,561.42**	**$1,189,326,612.81**	**$1,200,766,849.99**	**$3,805,392,024.21**

Figure 8 PivotTable connected to the Tabular model in DirectQuery mode

Once you complete your test, you can deploy the database to a Tabular server. This action will simply update the metadata, without performing any data import in memory. A process operation is not necessary in this case. All users will use the database in DirectQuery mode, regardless of the client they use (Excel, Power BI, or others).

Setting DirectQuery mode after deployment

If you have only one partition per table in your data model, you can enable and disable DirectQuery mode after deployment. Using an in-memory mode data model is a faster way to test your data, and is a good alternative to using sample partitions.

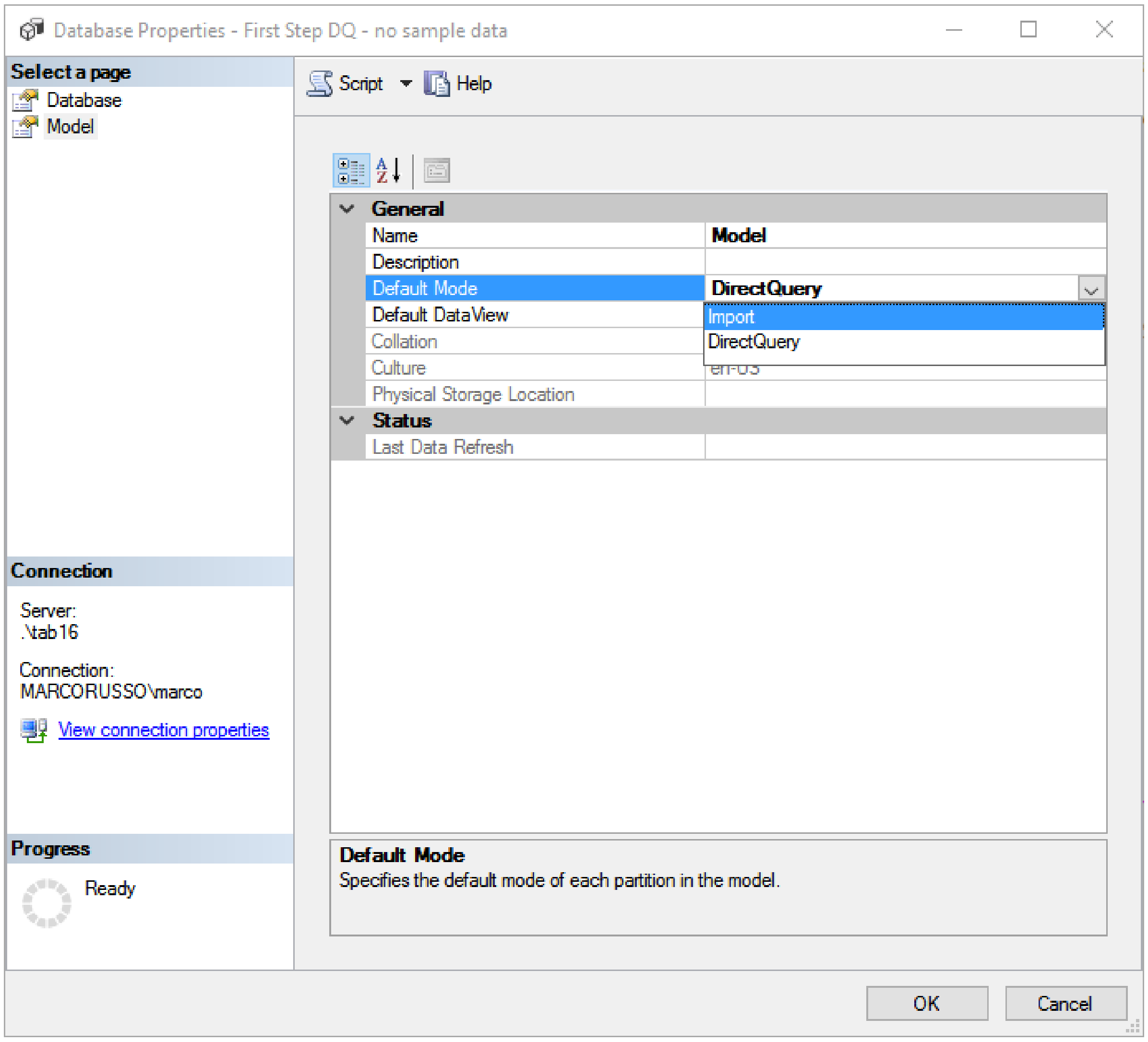
Knowing how to switch between DirectQuery and an in-memory model for testing your data gives you flexibility. For example, you can switch a Tabular in-memory model to DirectQuery, or you can switch a model originally created for DirectQuery to an in-memory model. In the latter case, any existing sample partitions must be removed in order to disable the DirectQuery mode.

In the next sections, we'll describe how to set DirectQuery after deployment using SSMS interactively, or by running a script.

Setting DirectQuery with SSMS

You can change the database's DirectQuery mode by opening the *Database Properties* dialog box: right-click the database name in the *Object Explorer* and choose the *Properties* menu item. Figure 9 shows the values available in the **Default Mode** property:

- **Import** — This corresponds to the DirectQuery Mode set to *Off* in SSDT. All tables are imported in memory and you must process the database in order to see data after you set this property.
- **DirectQuery** — This corresponds to the DirectQuery Mode set to *On* in SSDT. Any data loaded in memory for the tables is discharged, memory is freed, and all following queries will run in DirectQuery mode. There is no need to process the database after you enable DirectQuery mode.



Server:
.\tab16

Connection:
MARCORUSSO\marco

View connection properties

Ready

General	
Name	Model
Description	
Default Mode	DirectQuery
Default DataView	
Collation	
Culture	en-US
Physical Storage Location	
Status	
Last Data Refresh	

Import
DirectQuery

Default Mode
Specifies the default mode of each partition in the model.

Figure 9 PivotTable connected to the Tabular model in DirectQuery mode

Figure 9 shows a property called **Default DataView**. This setting defines the **Default DataView** property for partitions. Changing this property is not useful if you created partitions with SSDT, because those partitions always have the **DataView** option set to *Full* or *Sample*.

Setting DirectQuery with XMLA

If you want to change the DirectQuery Mode setting through an XMLA script, you can use the code described in the following example, where the **Default Mode** node can be either 0 or 1:

- 0: corresponds to **Default Mode** set to *Import* (DirectQuery Mode is *Off*)
- 1: corresponds to **Default Mode** set to *DirectQuery* (DirectQuery Mode is *On*)

```
<Alter xmlns="http://schemas.microsoft.com/analysisservices/2014/engine">
    <DatabaseID>First Step DQ - no sample data</DatabaseID>
    <Model>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:sql="urn:schemas-microsoft-com:xml-sql">
            <xs:element>
                <xs:complexType>
                    <xs:sequence>
                        <xs:element type="row"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:complexType name="row">
                <xs:sequence>
                    <xs:element name="DefaultMode" type="xs:long"
                                sql:field="DefaultMode" minOccurs="0"/>
                </xs:sequence>
            </xs:complexType>
        </xs:schema>
        <row xmlns="urn:schemas-microsoft-com:xml-analysis:rowset">
            <DefaultMode>1</DefaultMode>
        </row>
    </Model>
</Alter>
```

Setting DirectQuery with PowerShell

If you want to change the **DirectQuery Mode** setting through a PowerShell script, you can use the code described in the following example. Usually, you will put this code in a script to which you pass three parameters:

- SSAS instance name
- Database name
- Default Mode setting (either *Import* or *DirectQuery*)

```
param (
    [ValidateNotNullOrEmpty()][string]$ssasInstanceName,
    [ValidateNotNullOrEmpty()][string]$databaseName,
    [ValidateSet('Import','DirectQuery')][string]$defaultMode="" )
[System.Reflection.Assembly]::LoadWithPartialName(
    "Microsoft.AnalysisServices.Tabular")
$svr = new-Object Microsoft.AnalysisServices.Tabular.Server
$svr.Connect($ssasInstanceName)
$database = $svr.databases
$db = $database.GetByName($databaseName)
$db.Model.DefaultMode = $defaultMode
$db.Model.SaveChanges()
```

For example, you can set the DirectQuery mode for the **First Step DQ** database on the *TAB16* instance of the local server using the following command:

```
.\Set-DirectQueryMode.ps1 $ssasInstanceName'LOCALHOST\TAB16' $databaseName'First Step DQ'
$defaultMode'DirectQuery'
```

In order to revert to an in-memory mode data model, simply change the last parameter to Import:

```
.\Set-DirectQueryMode.ps1 $ssasInstanceName'LOCALHOST\TAB16' $databaseName'First Step DQ'
$defaultMode'Import'
```

Security setting in DirectQuery

When using DirectQuery, you can define security in two places. First, you can use the security roles defined in Analysis Services, just as you do with in-memory mode data models. As an alternative, you can apply security on the relational data source by instructing Analysis Services to impersonate the current user when it sends SQL queries to the data source. Usually, you choose either one technique or the other, but nothing prevents you from combining both techniques.

If you want to rely on the standard role-based security provided by Analysis Services, then you must be aware that all SQL queries will include the necessary predicates and joins to retrieve only the required data. Moreover, remember that, when using DirectQuery, there are restrictions to the DAX expressions that can be used in the role filters. These same restrictions apply to calculated columns.

If you already have row-level security implemented in the relational database and are also using Windows integrated security, you must configure Analysis Services to impersonate the current user, as described in the next section.

Security and impersonation with DirectQuery

A Tabular model in DirectQuery mode can connect to SQL Server in two ways:

- By always using the same user (defined in the connection).
- By impersonating the user that is querying Analysis Services.

The latter option requires Analysis Services to be configured for Kerberos constrained delegation, as explained at https://msdn.microsoft.com/en-us/library/dn194199.aspx. The following section focuses on how to configure the desired behavior on Analysis Services.

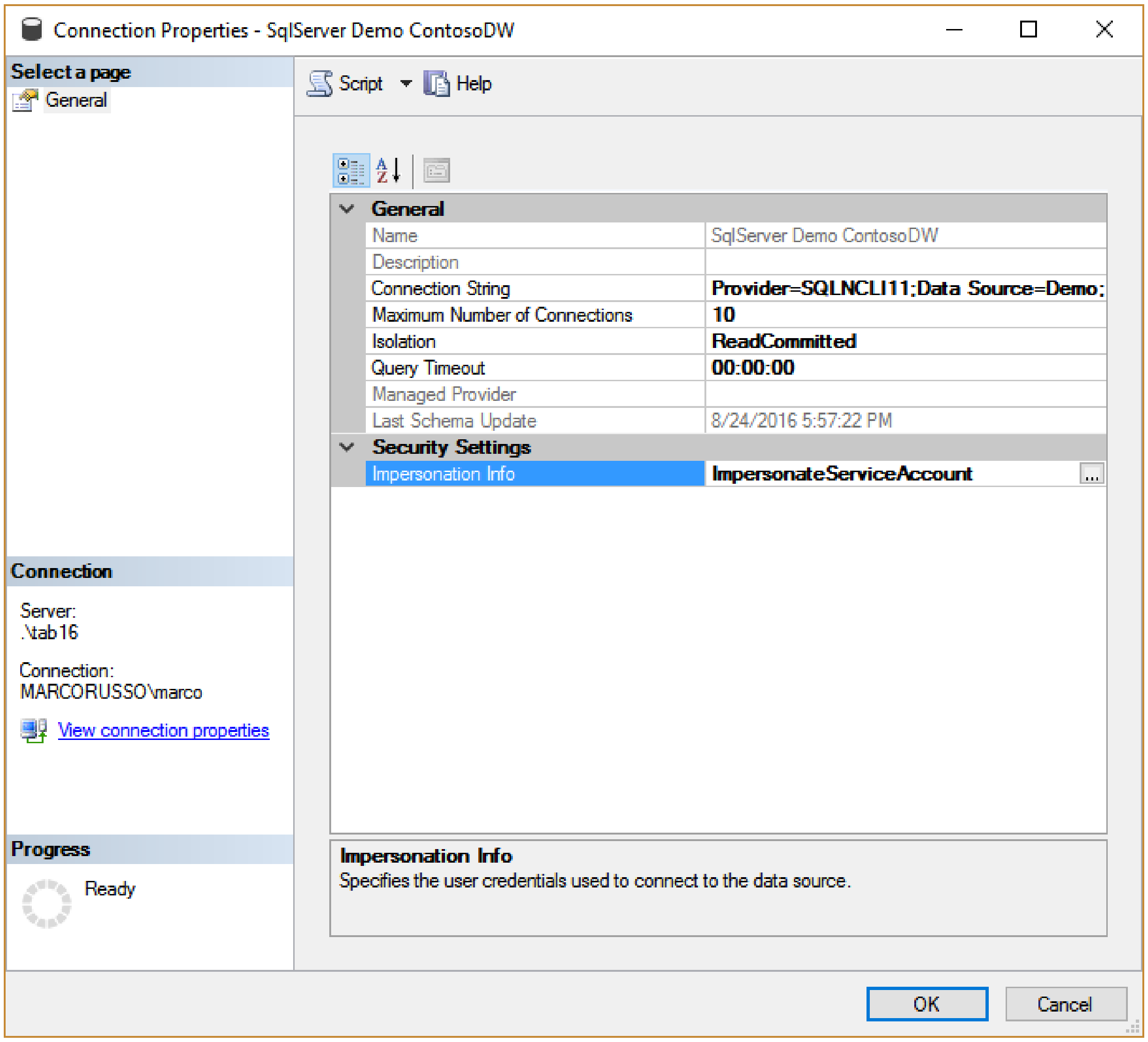
When you use DirectQuery, the model has a single connection, which has a particular configuration for impersonation (this is the property **impersonationMode** in the JSON file, but it is called **Impersonation Info** in SSMS, and just **Impersonation** in SSDT).

This security setting specifies the user that Analysis Services will impersonate when connecting to the data source. Impersonation determines which Windows user will execute the code in Analysis Services when connecting to the data source. If the source database supports Windows integrated security, the impersonated user will be shown as the user who is accessing and consuming data from the relational data source. If the source database does not support Windows integrated security, then this setting is not relevant.

For example, the source database could be a SQL Server database using Integrated Security. In this case, if Analysis Services impersonates the current user, SQL Server will receive queries from different users, and it could provide different results to the same requests depending on the user itself. If you have SQL Server 2016, this is a feature available with the row-level security on the relational database, as described at https://msdn.microsoft.com/en-us/library/dn765131.aspx.

To use impersonation:

- In the *Object Explorer* pane, right click on the database connection.
- In the *Connection Properties* dialog box, edit the SSMS connection properties (see Figure 10).
- In the Security Settings section, click on the drop-down menu to the right of *ImpersonateServiceAccount*.
- In the *Impersonation Information* dialog box, select *Use The Credentials Of The Current User* (see Figure 11).
- Click *OK* to close each dialog box.



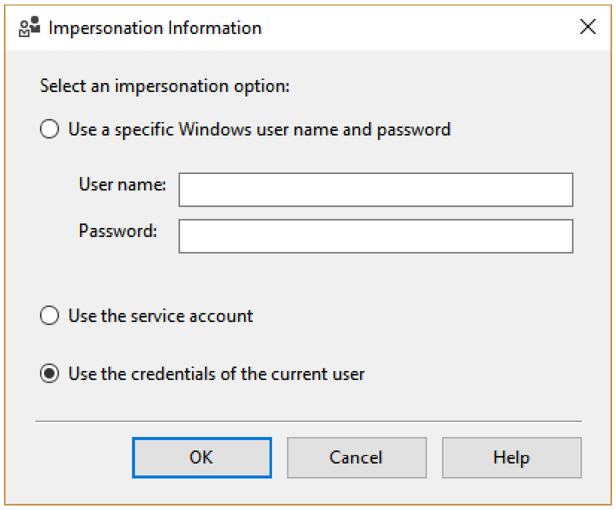
General	
Name	SqlServer Demo ContosoDW
Description	
Connection String	Provider=SQLNCLI11;Data Source=Demo;
Maximum Number of Connections	10
Isolation	ReadCommitted
Query Timeout	00:00:00
Managed Provider	
Last Schema Update	8/24/2016 5:57:22 PM
Security Settings	
Impersonation Info	ImpersonateServiceAccount

Impersonation Info
Specifies the user credentials used to connect to the data source.

Connection
Server:
.\tab16
Connection:
MARCORUSSO\marco
View connection properties

Progress
Ready

Figure 10 Impersonation Info in the Security Settings of the SSMS Connection Properties



Select an impersonation option:

( ) Use a specific Windows user name and password

User name:
Password:

( ) Use the service account

[✓ Use the credentials of the current user]

Figure 11 Options available in the Impersonation Information dialog box

Your Analysis Services instance will now use a different user for every connection made to SQL Server. It will impersonate the user connected to Analysis Services for each query. You can verify this behavior by checking the *NTUserName* column in SQL Profiler when monitoring the SQL queries received by SQL Server.

Using Row-level security on SQL Server earlier than 2016

If you are using a version of SQL Server earlier than SQL Server 2016, you can use an alternative technique for implementing row-level security based on permissions assigned to schemas. This method creates identical views for schemas that are used as a default by more than one user.

Steps to implement row-level security on SQL Server earlier than 2016:

These steps assume you are using the original model table created in the dbo schema (the default schema for every new SQL Server database). If you are using a different schema, use your schema name to replace "dbo" in the following instructions:

1. Define a schema on SQL Server for every user group.
2. Define a user on SQL Server to correspond to every user who you will enable on Analysis Services to access the Tabular model published in DirectQuery mode.
3. Assign to each user in SQL Server created in this way the corresponding schema (of the group to which he or she belongs) as a default schema. Each user belongs to a group. For each user, assign the group's schema as a default schema.
4. For each schema, grant **SELECT** permission to all the users in the group (that is, users who see the same rows of data).
5. For every table in the dbo schema that you reference in the Tabular model, create a SQL view with the same name in each schema. This view must include a `WHERE` condition that filters only the rows that should be visible to that group of users.
6. In the Tabular model, assign a SQL statement instead of a direct table binding to every *DirectQuery* partition, and remove any references to schemas in your SQL queries.

The result is:

- Analysis Services uses the user's credentials for SQL Server queries.
- The user has a default schema that uses views with the same name as the original tables or views.
- The schema has an additional `WHERE` condition that filters only the rows that the user is allowed to see.
- The SQL query generated by Analysis Services will use these tailored views to return only the rows that the user can see.
- To learn about another implementation of row-level security (based on a more dynamic, data-based approach), visit: http://sqlserverlst.codeplex.com.

Optimizing DirectQuery

As previously explained, optimizing DirectQuery involves tuning the relational data source to quickly answer the queries that are generated by DirectQuery. In order to optimize a DirectQuery model, you need to:

- Understand the workload imposed on the relational data source by your DAX code.
- Optimize the relational database for the specific workload.

Before using DirectQuery in your project:

- Take time to build a prototype.
- Gather the queries generated for your DAX formulas.
- Take time to understand the queries.

Understanding your queries and the workload your specific calculations generate will help you decide how to build an optimal relational data model.

The examples in the following sections use Microsoft SQL Server as the target data source, however the examples are also relevant for other supported relational databases.

Understanding datatype handling in DirectQuery

Our first example is a simple calculation to show how important it is to understand DirectQuery.

Imagine you have a fact table with a few billion rows. In the fact table, you have a simple **Quantity** column containing quantity sold of specific products. Since the quantities, taken individually, are small numbers, you store the **Quantity** column in SQL using an **INT**, which is a 32-bit integer column. Then, you build a simple DAX measure that computes the following:

```
SumOfQuantity := SUM ( Sales[Quantity] )
```

This measure works smoothly on a small fact table, but not on a large one. To compute the sum, the SSAS engine will send a SQL query like the following one:

```
SELECT SUM ( Sales.Quantity ) AS A0 FROM Sales
```

Because the Quantity column is an integer, SQL Server will use integer routines to compute the sum, which, on a large table, is likely to result in an overflow. In fact, on a large model this measure will usually fail to compute any result. (Note: when using in-memory mode (VertiPaq), the example data model and calculation above work just fine. In fact, in-memory mode data models usually store values using a simpler datatype system that is designed to avoid overflow errors. For example, in-memory mode's only data type for storing an integer is a 64-bit integer, while saving and storing smaller numbers is handled with compression techniques instead of using a smaller data type.)

SQL Server, on the other hand, has different data types to optimize storage, and uses math routines that are designed to provide the best performance using the columns of the tables it is scanning. This small semantic difference makes the system less useful, since you can't compute anything on top of it.

Options for how to handle the scenario described above:

- Store the **Quantity** column in SQL Server using a **BIGINT** column. This clearly violates some relational database best practices, because it involves using a large datatype to store a value that—in reality—doesn't need to be that large.
- Use a view to convert **Quantity** from **INT** to **BIGINT**. This avoids wasting storage space, but still shows the model a **BIGINT**.

The latter option may seem to be the most reasonable, but it includes a lot of complexity. The problem with using a view to perform datatype conversion is that the view will be called many times for each evaluation of the measure. With users browsing the model using Excel or Power BI, you can expect those evaluations to happen thousands of times a day. This imposes a heavy workload on SQL Server for performing datatype conversions.

The first option (using a **BIGINT** to store the quantity) removes the problem of datatype conversions at query time, and produces much better query plans. However, this model is not optimal from the SQL point of view.

Neither option is perfect. For SQL Server tables, we suggest using datatypes that are large enough to contain calculation results instead of the individual value, because this typically produces better query plans and a faster execution path. In order to lower the storage space required by SQL Server, you can always consider compression techniques and clustered columnstore indexes.

If you understand how DirectQuery works, then you can optimize your relational database setup for working with DirectQuery. Making correct decisions about things like which datatypes to use can save you a lot of time and iterations over the entire ETL process.

Simple query on a star schema and on snowflake schemas

This example shows SQL queries that are executed to resolve a simple DAX query that computes the sum of **Sales Amount**, slices it by **Calendar Year**, and adds a filter on a column. It uses two dimensions and one fact table in a basic star schema:

```
EVALUATE
CALCULATETABLE (
    ADDCOLUMNS (
        ALL ( 'Date'[Calendar Year] ),
        "Sales", [Sales Amount]
    ),
    Currency[Currency Code] = "USD"
)
```

This query generates two SQL queries:

- The first one simply retrieves the list of years.
- The second one returns the actual results.

```
SELECT TOP (1000001)
    [t0].[Calendar Year]
FROM
    ( SELECT * FROM Date ) AS [t0]
GROUP BY
    [t0].[Calendar Year]


SELECT TOP ( 1000001 )
    [t0].[Calendar Year],
    SUM ( [t1].[Line Amount] ) AS [a0]
FROM
    ( SELECT * FROM Sales ) AS [t1]
    LEFT OUTER JOIN ( SELECT * FROM Date ) AS [t0]
        ON [t1].[OrderDateKey] = [t0].[DateKey]
    LEFT OUTER JOIN ( SELECT * FROM Currency ) AS [t2]
        ON [t1].[CurrencyKey] = [t2].[CurrencyKey]
WHERE
    [t2].[Currency Code] = N'USD'
GROUP BY
    [t0].[Calendar Year]
```

Two queries are needed, because the engine needs to return years for which there are no sales, which are not included in the second query.

The **JOIN** type used in the queries depends on whether you created the relationships in the Tabular data model trusting referential integrity or not. Generally, it's always a good idea to use data models with referential integrity turned on, so that the `JOIN` becomes an `INNER JOIN`. This provides the engine with better options for defining the best execution plan.

Note: SSAS can assume that a relationship has a referential integrity when the property **relyOnReferentialIntegrity** is set to true in the Tabular model's relationship object. As of December 2016, SQL Server Data Tools (SSDT) does not support this property. You can edit the model.bim file adding the property **relyOnReferentialIntegrity** as you can see in the following code. Editing **relationships** in SSDT overrides the **relyOnReferentialIntegrity** setting, so you have to apply this setting again if you edit the relationship in SSDT. This problem will go away as soon as SSDT supports the **relyOnReferentialIntegrity** setting in the *Edit Relationship* dialog box.

```
    "relationships": [
      {
        "name": "84f32bea-9e5f-4c0b-9421-d8e6ef30ef10",
        "fromTable": "Sales",
        "fromColumn": "ProductKey",
        "toTable": "Product",
        "toColumn": "ProductKey",
        "relyOnReferentialIntegrity": true
      },
      {
          ...
```

Note: the engine's latest version implements redundant JOIN elimination, which aims to reduce the number of queries whenever possible. In fact, by rewriting the DAX query to take advantage of the new SUMMARIZECOLUMNS function, you can avoid the query that retrieves the years:

```
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS (
        'Date'[Calendar Year],
        "Sales", [Sales Amount]
    ),
    Currency[Currency Code] = "USD"
)
```

This behavior is also observed when running the same queries against the in-memory mode data model. There are no noticeable differences between the query executed against the in-memory mode storage engine and the ones against SQL.

If, instead of using a simple star schema, you have a snowflake schema (which has two or more dimensions related to each other), the query becomes slightly more complex with additional JOINs between dimensions. In the following example, instead of slicing by **Calendar Year**, which is an attribute of a dimension directly linked to the fact table, we slice by **Category**, which is a dimension related to **Product Subcategory**, and finally to **Product**, which, in turn, relates to the **Sales** fact table:

```
EVALUATE
CALCULATETABLE (
    ADDCOLUMNS (
        ALL ( 'Product Category'[Category] ),
        "Sales", [Sales Amount]
    ),
    Currency[Currency Code] = "USD"
)
```

The pattern is very similar to the previous one. The difference is additional JOINs with **Product Subcategory** and **Product Category**, as shown in the following SQL code:

```
SELECT TOP (1000001)
    [t6].[Category],
    SUM([t1].[Line Amount]) AS [a0]
FROM
    (SELECT * FROM Sales ) AS [t1]
    LEFT OUTER JOIN (SELECT * FROM Currency ) AS [t2]
        ON ([t1].[CurrencyKey] = [t2].[CurrencyKey])
    LEFT OUTER JOIN (SELECT * FROM Product ) AS [t5]
        ON ([t1].[ProductKey] = [t5].[ProductKey])
    LEFT OUTER JOIN (SELECT * FROM [Product Subcategory] ) AS [t7]
        ON ([t5].[ProductSubcategoryKey] = [t7].[ProductSubcategoryKey])
    LEFT OUTER JOIN (SELECT * FROM [Product Category] ) AS [t6]
        ON ([t7].[ProductCategoryKey] = [t6].[ProductCategoryKey])
WHERE
    [t2].[Currency Code] = N'USD'
GROUP BY
    [t6].[Category]
```

As you can see from these examples, a good SQL database should answer these queries quickly and, in order to optimize its behavior, you can simply follow the well-known best practices for data warehouse modeling in a relational database.

Filter over a calculated column

As described in previous examples, applying a filter to a column that directly maps to a SQL column results in the filter being applied to the SQL database column as well. It's interesting to see what happens if—instead of filtering a physical column—you apply the filter to a calculated column. In DAX, a calculated column is not different from a physical one, but we learned in the previous sections that calculated columns in DirectQuery mode are handled by injecting the SQL code that computes the column in the query.

Thus, you define a calculated column with the following code:

```
Sales[Calc Line Amount] = Sales[Net Price] * Sales[Quantity]
```

Then, you can apply a filter to the column in a query:

```
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS ( 'Date'[Calendar Year], "Amount", [Sales Amount] ),
    Sales[Calc Line Amount] > 100,
    Currency[Currency Code] = "USD"
)
```

Here is the resulting SQL code:

```
SELECT TOP (1000001)
    [t0].[Calendar Year],
    SUM([t1].[Line Amount]) AS [a0]
FROM
    (SELECT
        [t1].[CurrencyKey] AS [CurrencyKey],
        [t1].[OrderDateKey] AS [OrderDateKey],
        [t1].[Line Amount] AS [Line Amount],
        [t1].[Net Price] * [t1].[Quantity] AS [Calc Line Amount]
     FROM
        (SELECT * FROM Sales ) AS [t1]
    ) AS [t1]
    LEFT OUTER JOIN (SELECT * FROM Date ) AS [t0]
        ON [t1].[OrderDateKey] = [t0].[DateKey]
    LEFT OUTER JOIN (SELECT * FROM Currency ) AS [t2]
        ON [t1].[CurrencyKey] = [t2].[CurrencyKey]
WHERE
    [t2].[Currency Code] = N'USD'
    AND [t1].[Calc Line Amount] > CAST(N'100' AS MONEY)
GROUP BY
    [t0].[Calendar Year];
```

The SQL Server optimizer places the filter as near as possible to the scanning of the fact table. You can see this behavior in a model implemented using clustered columnstore indexes by looking at the following excerpt from the query plan:

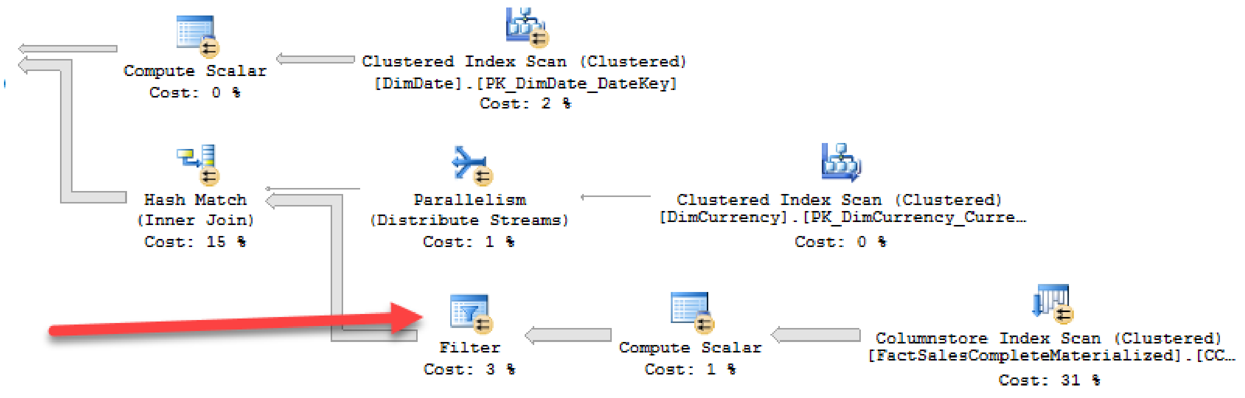

Compute Scalar
Cost: 0 %
Clustered Index Scan (Clustered)
[DimDate].[PK_DimDate_DateKey]
Cost: 2 %
Hash Match
(Inner Join)
Cost: 15 %
Parallelism
(Distribute Streams)
Cost: 1 %
Clustered Index Scan (Clustered)
[DimCurrency].[PK_DimCurrency_Curre…
Cost: 0 %
Filter
Cost: 3 %
Compute Scalar
Cost: 1 %
Columnstore Index Scan (Clustered)
[FactSalesCompleteMaterialized].[CC…
Cost: 31 %

Figure 12 Filtering occurs near the fact table

Nevertheless, since the filter is made on an expression instead of a physical column, the SQL engine needs to scan the entire fact table to retrieve the rows satisfying the condition. The output list of the bottom **Columnstore Index Scan** contains all the columns that are needed to complete the query, including the keys needed to perform the join with **Currency** and **Date**, and the columns needed to calculate the filter condition (which is a multiplication of two physical columns: **Quantity** and **Net Price**).

The engine scans the columnstore, which retrieves all needed columns so that it can further apply the filter and use the columns for subsequent joins. This behavior is very different from what you see with in-memory mode.

In fact, if you execute the same query in in-memory mode, the result is computed by this **xmSQL** query:

```
WITH
      $Expr0 := 'Sales'[Net Price] * 'Sales'[Quantity]
SELECT
    'Date'[Calendar Year],
    SUM ( @$Expr0 ) AS $Measure0
FROM 'Sales'
    LEFT OUTER JOIN 'Date' ON 'Sales'[OrderDateKey]='Date'[DateKey]
    LEFT OUTER JOIN 'Currency' ON 'Sales'[CurrencyKey]='Currency'[CurrencyKey]
WHERE
      'Currency'[Currency Code] = 'USD' VAND
      'Sales'[Calc Line Amount] > 100
```

The main difference, here, is that the in-memory mode engine scans the fact table and the filter is applied to a physical column, instead of an expression. Moreover, the join between the dimensions and the fact table is executed straight into the in-memory mode engine, without the need to return the columns involved in the join. Moreover, since the filter is on a physical column, fewer rows are actually scanned.

Let's summarize the topics covered so far regarding filters over a calculated column:

- The expression of the calculated column is evaluated every time the query is executed, for the entire table. Thus, complex calculations might become expensive.
- The full table needs to be scanned in order to compute the calculated column, no matter how restrictive the filter is.
- All columns needed for further joins are returned from the scan of the fact table, because the filtering for other dimensions happens later.

Of course, the cost of scanning the fact table is somewhat constant, whereas the additional cost of performing joins and evaluating the rest of the query strictly depends on how restrictive the filter is. For example, we executed the previous query changing the filter value from 100 to 100,000 in four steps, and we executed the same query with no filter on the column. The higher the value of the parameter, the smaller the number of rows surviving the filter. Moreover, we tested the same query with the calculated column and with a column containing the same values, but previously computed and stored, so that it became a physical column. The results are interesting, as you can see in the following table (time is in milliseconds):

Filter	Calculated Column	Physical Column	Perf
No Filter	202	190	1.06x
100	140	87	1.61x
1,000	92	33	2.78x
10,000	75	10	7.50x
100,000	73	10	7.30x

As you can see, filtering a physical column is always faster than filtering a calculated column, and, the more restrictive the filter is, the bigger the performance gain. The fixed amount of time needed to scan the fact table, which you always pay in the case of a calculated column, becomes a more important addendum in the more restrictive filter.

In conclusion, calculated columns are fine for computing results (as it is very often the case), but they have a much higher performance price when you layer a filter on top of them. In such a case, pre-computing the calculated column in the relational database becomes a much better alternative even if it increases the storage needs.

Using time intelligence functions with additive measures

Among the many filtering techniques, time intelligence plays an important role, because it is present in nearly any analytical solution. Thus, it's important to understand how the engine is going to perform when using time intelligence calculations. Moreover, this example let's us better analyze some of the optimizations that are present in the engine regarding additive calculations.

This simple example uses the `SAMEPERIODLASTYEAR` function:

```
DEFINE
    MEASURE Sales[Sales PY] =
        CALCULATE ( [Sales Amount], SAMEPERIODLASTYEAR ( 'Date'[Date] ) )
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS (
        'Date'[Calendar Year Month],
        'Date'[Calendar Year Month Number],
        "Sales", [Sales Amount],
        "Sales PY", [Sales PY]
    ),
    'Date'[Calendar Year] = "CY 2008"
)
ORDER BY 'Date'[Calendar Year Month Number]
```

The query returns **Sales** and **Sales in the previous year**, sliced by month, for calendar year 2008. In total, it returns 12 rows. As soon as you start using time intelligence functions, the query becomes much more complex both to analyze and to understand; yet it provides good insights into how the DirectQuery engine processes the query. In fact, the previous query generates four different SQL queries, two of which have some relevance. The first one retrieves **Sales** *at the day granularity* with a very long list of dates as a filter, which includes the entire year 2007. The second one is identical, but it retrieves the year 2008.

```
SELECT TOP (1000001)
    [t0].[Date],
    SUM ( [t1].[Line Amount] ) AS [a0]
FROM
    (SELECT * FROM Sales ) AS [t1]
    LEFT OUTER JOIN (SELECT * FROM Date ) AS [t0]
        ON [t1].[OrderDateKey] = [t0].[DateKey]
WHERE
    [t0].[Date] IN (
        CAST(N'2007-01-19 00:00:00' AS DATETIME),
        CAST(N'2007-09-02 00:00:00' AS DATETIME),
        ...
        ...
        ...
        CAST(N'2008-12-07 00:00:00' AS DATETIME),
        CAST(N'2008-04-20 00:00:00' AS DATETIME)
    )
GROUP BY
    [t0].[Date]
```

Note:

- The list of dates is provided as a filter straight in the query, using `CAST` from a string. This generates queries that are very long and, of course, require some time to be parsed and optimized. The length of the query basically depends on the number of days included in the full query.
- The filter in `CALCULATETABLE` filters the year 2008, but, due to the presence of `SAMEPERIODLASTYEAR`, the list of dates in one of the queries includes the entire year 2007. In fact, we have two queries, one per year.
- `SUMMARIZECOLUMNS` requested values grouped by month, but the query retrieves data at the day level.

Why is the SQL query so different from the original DAX one? Because the engine detected that the measure to compute is an additive one. In fact, the calculation only requires a `SUM`, which results in a completely additive calculation. In other words, the engine knows that it can compute the sum of one month by summing the individual dates of that month. This assumption would be incorrect if you use a non-additive measure like a distinct count. We will examine this scenario later, after discussing additive measures, which are the most common ones.

The engine retrieved the value of the measure at the day granularity for two years, 2007 and 2008, with two different SQL queries. This is what it needs from the storage engine. The formula engine, later, will aggregate days in months in both 2007 and 2008, producing the result that includes, in January 2008, the sales of the same period in the previous year (which is January 2007).

This is interesting behavior, because it shows that the engine analyzes the additivity of the measure and generates queries, which are smart in gathering data at a different granularity than the one requested, performing the next step of aggregation in the formula engine.

Other SQL queries can fully complete the DAX query resolution, but they are less interesting.

The important point is that if you use time intelligence functions with additive measures, the engine retrieved 365 rows from SQL Server to produce a result with 12 rows for the **Sales PY** measure. This behavior (going at the most granular level of the **Date** table) seems to happen only with the predefined time intelligence functions. In fact, if you redefine the measure using a custom time intelligence calculation that relies only on standard DAX functions, the resulting SQL queries are different. For example, consider the following DAX query, which works only at the month level and does not rely on predefined time intelligence functions:

```
DEFINE
    MEASURE Sales[Sales PY] =
        SUMX (
            VALUES ( 'Date'[Calendar Year Month Number] ),
            CALCULATE (
                [Sales Amount],
                ALL ( 'Date' ),
                FILTER (
                    ALL ( 'Date'[Calendar Year Month Number] ),
                    'Date'[Calendar Year Month Number]
                        = EARLIER ( 'Date'[Calendar Year Month Number] ) - 100
                )
            )
        )
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS (
        'Date'[Calendar Year Month],
        'Date'[Calendar Year Month Number],
        "Sales", [Sales Amount],
        "Sales PY", [Sales PY]
    ),
    'Date'[Calendar Year] = "CY 2008"
)
ORDER BY 'Date'[Calendar Year Month Number]
```

Despite the code being more complex, the engine still detects that the measure is an additive one, and uses the optimization of gathering, with two queries, values in years 2007 and 2008, performing a further join between the two sets later in the formula engine.

This time, the SQL code for sales in year 2017 is the following one:

```
SELECT TOP (1000001)
    [t0].[Calendar Year Month Number],
    SUM ( [t1].[Line Amount] ) AS [a0]
FROM
    (SELECT * FROM Sales ) AS [t1]
    LEFT OUTER JOIN (SELECT * FROM Date ) AS [t0]
        ON [t1].[OrderDateKey] = [t0].[DateKey]
WHERE [t0].[Calendar Year Month Number] IN (
    200707, 200708, 200712, 200701, 200702, 200705,
    200709, 200710, 200711, 200703, 200704, 200706
)
GROUP BY
    [t0].[Date]
```

As you can see, this time the query returns only 12 rows for year 2017. Thus, it retrieves 24 rows to produce a result containing twelve rows (the query for 2018 is similar in both cases, because the **Sales** measure does not use any time intelligence function). This latter query looks more optimized than the previous one with the potential issue that the values computed might not be useful to perform different levels of aggregation. In fact, if the engine obtains the values at the most granular level (which is the date in this case), it will be able to use the result to perform any aggregation at a lower granularity, which it cannot do if the values are returned at the month granularity.

Using time intelligence functions with non-additive measures

As you have seen in the previous example, if the engine detects that a measure is an additive one, then it gathers data at the key granularity and performs the aggregation in the formula engine, which looks like a very smart plan. What happens if we force it to re-compute the value by using a non-additive measure like, for example, a distinct count?

In such a case, the engine cannot rely on data at the most granular level, so it needs to run one query for each row of the result. You can use both standard and custom time intelligence functions. The following query uses the standard time intelligence function SAMEPERIODLASTYEAR:

```
DEFINE
    MEASURE Sales[Customers] =
        DISTINCTCOUNT ( Sales[CustomerKey] )
    MEASURE Sales[Customers PY] =
        CALCULATE ( [Customers], SAMEPERIODLASTYEAR ( 'Date'[Date] ) )
EVALUATE
SUMMARIZECOLUMNS (
    'Date'[Calendar Year Number],
    "Customers", [Customers],
    "Customers PY", [Customers PY]
)
ORDER BY 'Date'[Calendar Year Number]
```

With a custom time intelligence calculation you can limit the granularity of the query at the year level:

```
DEFINE
    MEASURE Sales[Customers] =
        DISTINCTCOUNT ( Sales[CustomerKey] )
    MEASURE Sales[Customers PY] =
        CALCULATE (
            [Customers],
            ALL ( 'Date'[Calendar Year Number] ),
            VAR PrevYear =
                VALUES ( 'Date'[Calendar Year Number] ) - 1
            RETURN
                FILTER (
                    ALL ( 'Date'[Calendar Year Number] ),
                    'Date'[Calendar Year Number] = PrevYear
                )
        )
EVALUATE
SUMMARIZECOLUMNS (
    'Date'[Calendar Year Number],
    "Customers", [Customers],
    "Customers PY", [Customers PY]
)
ORDER BY 'Date'[Calendar Year Number]
```

In both cases, the presence of DISTINCTCOUNT, which is non-additive, forces the engine to execute multiple queries. For this latter DAX query, there are five SQL queries, one for each year, all of which have the same format:

```
SELECT
    COUNT_BIG ( DISTINCT [t1].[CustomerKey] )
      + MAX (
          CASE WHEN [t1].[CustomerKey] IS NULL THEN 1
              ELSE 0
        END
      ) AS [a0]
FROM
    (SELECT * FROM Sales ) AS [t1]
    LEFT OUTER JOIN ( SELECT * FROM Date ) AS [t0]
        ON [t1].[OrderDateKey] = [t0].[DateKey]
WHERE
    [t0].[Calendar Year Number] IN (2007)
    AND [t0].[Calendar Year Number] IN (2006, 2010, 2005, 2009, 2007, 2008)
```

In this case, DirectQuery is similar to in-memory mode (VertiPaq), because both engines require executing multiple scans of the fact table to gather the distinct counts. If the fact table is on a clustered columnstore index, then performance is quite good in SQL Server, too. This kind of calculation becomes more expensive when the number of rows included in the result is very high, because of the high number of SQL queries executed (one for each row of the result).

Using time intelligence functions with semi-additive measures

In DAX, you typically use one of two patterns for semi-additive measures: LASTDATE and LASTNONBLANK. LASTDATE always returns the last date of the set, and it might return blank values for an incomplete month, whereas LASTNONEMPTY always returns data for the last date for which there is a value to show. It's well known that LASTNONEMPTY is heavier than LASTDATE. How does DirectQuery perform with these two patterns?

The following example shows a query that computes the LASTDATE pattern:

```
DEFINE
    MEASURE Sales[Sales Last Day] =
        CALCULATE ( [Sales Amount], LASTDATE ( 'Date'[Date] ) )
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS (
        'Date'[Calendar Year Month Number],
        "Sales", [Sales Amount],
        "Sales Last Day", [Sales Last Day]
    ),
    'Date'[Calendar Year Number] = 2008
)
ORDER BY 'Date'[Calendar Year Month Number]
```

In this case, the SQL code generated is very efficient, since the `LASTDATE` is computed before sending the query to SQL, so that the scan of the fact table only retrieves data for the last day of the month in the selection:

```
SELECT TOP (1000001)
    [t0].[Date],
    SUM ( [t1].[Line Amount] ) AS [a0]
FROM
    (SELECT * FROM Sales ) AS [t1]
    LEFT OUTER JOIN (SELECT * Date ) AS [t0]
        ON [t1].[OrderDateKey] = [t0].[DateKey]
WHERE
    [t0].[Date] IN (
        CAST(N'2008-06-30 00:00:00' AS DATETIME),
        CAST(N'2008-08-31 00:00:00' AS DATETIME),
        CAST(N'2008-03-31 00:00:00' AS DATETIME),
        CAST(N'2008-04-30 00:00:00' AS DATETIME),
        CAST(N'2008-01-31 00:00:00' AS DATETIME),
        CAST(N'2008-02-29 00:00:00' AS DATETIME),
        CAST(N'2008-05-31 00:00:00' AS DATETIME),
        CAST(N'2008-09-30 00:00:00' AS DATETIME),
        CAST(N'2008-10-31 00:00:00' AS DATETIME),
        CAST(N'2008-07-31 00:00:00' AS DATETIME),
        CAST(N'2008-11-30 00:00:00' AS DATETIME),
        CAST(N'2008-12-31 00:00:00' AS DATETIME)
    )
GROUP BY
    [t0].[Date];
```

When using `LASTNONBLANK`, on the other hand, you need to pay attention to small details, to speed up the execution of the query. When using in-memory mode, you typically use a SUM to check for blanks. For example, in the following code we put **[Sales Amount]** to check for emptiness.

```
DEFINE
    MEASURE Sales[Sales Last Day] =
        CALCULATE (
            [Sales Amount],
            LASTNONBLANK ( 'Date'[Date], [Sales Amount] )
        )
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS (
        'Date'[Calendar Year Month Number],
        "Sales", [Sales Amount],
        "Sales Last Day", [Sales Last Day]
    ),
    'Date'[Calendar Year Number] = 2008
)
ORDER BY 'Date'[Calendar Year Month Number]
```

The engine needs to scan the fact table and compute the measure first in order to gather the last date for which the measure is not blank. This results in an additional query returning the measure's value (`Sum` of **Line Amount**), which is then evaluated by the formula engine to determine which of the many dates returned is the last non blank:

```
SELECT TOP ( 1000001 )
        [t0].[Date],
        SUM ( [t1].[Line Amount] ) AS [a0]
FROM    ( SELECT * FROM Sales ) AS [t1]
        LEFT OUTER JOIN ( * FROM Date ) AS [t0]
            ON [t1].[OrderDateKey] = [t0].[DateKey]

WHERE
    [t0].[Date] IN (
        CAST(N'2008-03-15 00:00:00' AS DATETIME),
        CAST(N'2008-02-07 00:00:00' AS DATETIME),
        CAST(N'2008-06-20 00:00:00' AS DATETIME),
        --
        -- all days for year 2008 included here
        --
        CAST(N'2008-07-27 00:00:00' AS DATETIME),
        CAST(N'2008-12-25 00:00:00' AS DATETIME)
    )
GROUP BY [t0].[Date]
```

This latter SQL query is heavy, because it needs to scan two columns (**Date** and **Line Amount**) along the entire fact table, after having applied a filter to the date column. By rewriting the DAX code to avoid using the measure and leveraging, and instead use the `ISEMPTY` function, you obtain the following pattern:

```
DEFINE
    MEASURE Sales[Sales Last Day] =
        CALCULATE (
            [Sales Amount],
            LASTNONBLANK ( 'Date'[Date], CALCULATE ( NOT ISEMPTY ( Sales ) ) )
        )
EVALUATE
CALCULATETABLE (
    SUMMARIZECOLUMNS (
        'Date'[Calendar Year Month Number],
        "Sales", [Sales Amount],
        "Sales Last Day", [Sales Last Day]
    ),
    'Date'[Calendar Year Number] = 2008
)
ORDER BY 'Date'[Calendar Year Month Number]
```

Now, instead of using **[Sales Amount]** to check for BLANK, we use `NOT ISEMPTY`, which results in the following SQL code:

```
SELECT TOP (1000001)
    [t0].[Date]
FROM
    (SELECT * FROM Sales ) AS [t1]
    LEFT OUTER JOIN (SELECT * FROM Date ) AS [t0]
        ON ([t1].[OrderDateKey] = [t0].[DateKey])
WHERE
    [t0].[Date] IN (
        CAST(N'2008-03-15 00:00:00' AS DATETIME),
        CAST(N'2008-02-07 00:00:00' AS DATETIME),
        CAST(N'2008-06-20 00:00:00' AS DATETIME),
        --
        -- all days for year 2008 included here
        --
        CAST(N'2008-07-27 00:00:00' AS DATETIME),
        CAST(N'2008-12-25 00:00:00' AS DATETIME)
    )
GROUP BY
    [t0].[Date]
```

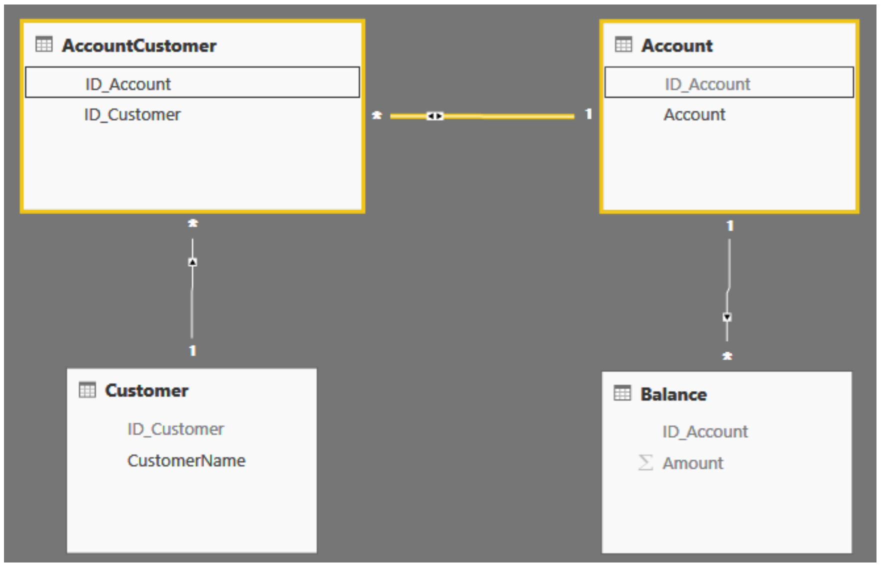
This query is faster and easier to optimize in SQL Server, because it uses only the **Date** column and does not need to scan **Line Amount**. This query relies on the existing relationship between **Date** and **Sales**, producing a better execution plan.

As expected, when using a DirectQuery model, you need to pay attention to details in the DAX code in order to let the engine generate queries that can be better optimized in SQL.

Many-to-many relationships

In order to test the behavior of many-to-many relationships, we use a different model that contains balances of accounts. Each account can be owned by multiple customers and, at the same time, each customer can have multiple accounts.

From the data modeling point of view, this results in a bridge table between accounts and customers, as in the following model:



The relationship between the bridge table and the account table is set as bidirectional, in order to make the filter flow smoothly from **Customer** to **Balance**. The resulting measure is, of course, a non-additive one, as per the nature of many-to-many relationships.

In this scenario, a simple query like the following one results in a complex SQL query, because the filter needs to be propagated from **Customer** to **AccountCustomer**, then to **Account**, and finally to **Balance**:

```
EVALUATE
SUMMARIZECOLUMNS ( Customer[CustomerName], "Amount", [Sum Amount] )
```

In fact, this simple query is translated into the following SQL code:

```
SELECT TOP (1000001)
    [semijoin1].[c13],
    SUM ( [a0] ) AS [a0]
FROM
    (SELECT
        [t0].[ID_Account] AS [c1],
        SUM ( [t2].[Amount] ) AS [a0]
     FROM
        (SELECT * FROM Balance ) AS [t2]
        LEFT OUTER JOIN (SELECT * FROM Account ) AS [t0]
            ON [t2].[ID_Account] = [t0].[ID_Account]
     GROUP BY
        [t0].[ID_Account]
    ) AS [basetable0]
    INNER JOIN (SELECT
                    [t0].[ID_Account] AS [c1],
                    [t3].[CustomerName] AS [c13]
                FROM
                    (SELECT * FROM AccountCustomer ) AS [t1]
                    LEFT OUTER JOIN (SELECT * FROM Account ) AS [t0]
                        ON [t1].[ID_Account] = [t0].[ID_Account]
                    LEFT OUTER JOIN (SELECT * FROM Customer ) AS [t3]
                        ON [t1].[ID_Customer] = [t3].[ID_Customer]
                GROUP BY
                    [t0].[ID_Account],
                    [t3].[CustomerName]
               ) AS [semijoin1]
        ON [semijoin1].[c1] = [basetable0].[c1]
            OR [semijoin1].[c1] IS NULL
            AND [basetable0].[c1] IS NULL
GROUP BY
    [semijoin1].[c13]
```

The query is somewhat long, but you can easily split it into two simpler subqueries:

1. The first part selects the sum of amount split by account
2. The second part returns the customer names and their accounts

By joining the two subqueries, you obtain the sum of balance for all the accounts of a given customer, which is the desired result.

The query, by itself, is not complex and is reasonable given the requirement. Nevertheless, it’s important to note that query complexity increases as soon as you mark some of the relationships as bidirectional. This is because when you have a bidirectional relationship, the model becomes non-additive whenever you apply a filter that traverses the bidirectional relationship. When using in-memory mode, the scenario is similar. Bidirectional filtering comes with a lot of power, but also a lot of complexity.

On small databases, the query is expected to be very fast. On larger models, the bridge table size (which drives the complexity of the second subquery of the full one) might be an issue. Again, the goal here is not to discuss performance in detail, but instead to give you an idea of the kind of queries that are generated to resolve DAX code in DirectQuery.

Comparing DirectQuery with in-memory mode (VertiPaq)

In the previous sections, we analyzed several queries in order to give insight into how the DirectQuery engine works. Of course, real-world situations are more complex, and we recommend conducting a complete analysis of your system using our examples for reference.

It's important to understand how DirectQuery and in-memory mode compare in terms of performance.

To test this, we created a database with a fact table containing 4 billion rows and executed several queries against the same model in both DirectQuery mode and in-memory mode. The fact table is stored on a clustered columnstore in SQL Server, and the columns have the correct datatypes, in order to avoid any interference coming from datatype casting. Also, the tests were executed on the same machine.

Disclaimer: These tests were not executed in a perfect environment and not all possible scenarios were executed, so your results may vary. However, these numbers may help set performance expectations for DirectQuery.

- General: We noticed that DirectQuery runs two to three times slower than in-memory mode. Depending on the individual query and the number of joins, performance degradation because of the adoption of DirectQuery varied from 1.5x to 2.9x. DirectQuery was never faster than in-memory mode, which is expected.
- Bidirectional filtering has a huge cost. All queries with bidirectional filtering activated (either in the code or in the model) resulted in slower performance. The DirectQuery queries executed in a range between 3x and 30x, depending on the complexity of the query.
- Referential Integrity is very important. In most of the queries, we were able to make them faster by simply stating for the relationships that the engine can trust referential integrity. This is because the engine uses `INNER JOIN` instead of `LEFT OUTER JOIN`, giving the SQL optimizer many more options to move the filters closer to the fact table, reducing the number of rows it needs to process.

Conclusion

If you lived in a perfect world where the relational database is fast enough to provide a result to any query in less than one second, the choice about whether to use DirectQuery or not would be an easy one. By using DirectQuery, there would be no need to copy and process data on an in-memory mode engine (VertiPaq), Analysis Services would be just a semantic layer on top of the relational database, and the performance would be guaranteed by the server running the only copy of the data. Zero processing time, zero latency, data always up-to-date, and less memory needed for Analysis Services.

Unfortunately, reality is far from perfect. Even using a columnstore index on SQL Server, which is based on the same technology as an in-memory mode data model, you won't achieve the same performance that you can get with an in-memory model in Analysis Services. Thus, choosing between DirectQuery and in-memory mode is a matter of tradeoffs and priorities.

The main reason to use DirectQuery is to reduce the latency between updates on the relational database and availability of the same data in the Tabular model. DirectQuery removes this latency, but it uses the same relational engine used to update data. This means that the same relational database will manage concurrent updates and queries, so you should figure out whether the server will be able to support the concurrent workload. Using in-memory mode is also a way to reduce the pressure on a relational database, because an increasing number of users does not change the workload on the data source. If you import data using in-memory mode, you read data only once, regardless of the following queries run by the users. Moreover, keep in mind that removing latency (going real-time) brings the problem of report coherency, which might be difficult to explain to the users.

Another possible scenario for using DirectQuery is when you have a database that is too large to fit in the memory of a single Analysis Services server. For example, tables with more than 100 billion rows might be hard to manage with in-memory mode, so if you cannot reduce the cardinality of data loaded in memory, you might use particular architectures to handle the volume, for example by using Microsoft SQL Server Analytics Platform System (APS), or the Microsoft Azure SQL Data Warehouse. In these conditions, you can use an Analysis Services Tabular model in DirectQuery mode to create a semantic layer to access a very large database. You might not expect the same level of performance you are used to in an interactive scenario, but this approach could be faster than any other available option, even if certain queries will still require many seconds to complete, if not minutes.

Finally, here are the two main scenarios where you might consider using DirectQuery:

- **A small database that is updated frequently:** the concept of "small" depends on the performance and optimization of the relational database. For example, a SQL Server database using clustered columnstore indexes can handle much more data than a database based on classical indexes.
- **A large database that would not fit in memory:** even if this scenario does not guarantee an interactive user experience navigating data, as you might expect by most of the in-memory mode models you can create in Tabular, it could be good enough to support scenarios where a query can run in many seconds if not minutes. The semantic value provided by Tabular is an important added value for the analytical solution.

In SQL 2016, DirectQuery is a useful option for some very specific needs. It is not meant to solve all issues related to complex analytical systems, and it requires a deep knowledge of its internals. It's not an in-memory mode engine replacement, but will provide the foundation of many real-time analytical systems.

More information

http://www.sqlbi.com/: SQLBI Web site

http://www.microsoft.com/sqlserver/: SQL Server Web site

http://technet.microsoft.com/en-us/sqlserver/: SQL Server TechCenter

http://msdn.microsoft.com/en-us/sqlserver/: SQL Server DevCenter

https://msdn.microsoft.com/en-us/library/gg413422.aspx DAX Reference

Appendix A: Semantic differences in DAX

This appendix lists the types of semantic differences that you can expect, and describes any limitations that might apply to the usage of functions or to query results. An updated list of these differences is available at https://msdn.microsoft.com/en-us/library/mt723603.aspx.

Comparisons

DAX in in-memory models support comparisons of two expressions that resolve to scalar values of different data types. However, models that are deployed in DirectQuery mode use the data types and comparison operators of the relational engine, and therefore might return different results.

The following comparisons will always return an error when used in a calculation on a DirectQuery data source:

- Numeric data type compared to any string data type
- Numeric data type compared to a Boolean value
- Any string data type compared to a Boolean value

In general, DAX is more forgiving of data type mismatches in in-memory models and will attempt an implicit cast of values up to two times, as described in this section. However, formulas sent to a relational data store in DirectQuery mode are evaluated more strictly, following the rules of the relational engine, and are more likely to fail.

Comparisons of strings and numbers

```
EXAMPLE: “2” < 3
```

The formula compares a text string to a number. The expression is true in both DirectQuery mode and in-memory models.

In an in-memory model, the result is true because numbers as strings are implicitly cast to a numerical data type for comparisons with other numbers. SQL also implicitly casts text numbers as numbers for comparison to numerical data types.

This represents a change in behavior from the first version of PowerPivot, which would return *false*, because the text *“2”* would always be considered larger than any number.

Comparison of text with Boolean

```
EXAMPLE: “VERDADERO” = TRUE
```

This expression compares a text string with a Boolean value. In general, for DirectQuery or in-memory models, comparing a string value to a Boolean value results in an error. The only exceptions to the rule are when the string contains the word *true* or the word *false*; if the string contains any *true* or *false* values, a conversion to Boolean is made and the comparison takes place giving the logical result.

Comparison of nulls

```
EXAMPLE: EVALUATE ROW("X", BLANK() = BLANK())
```

This formula compares the SQL equivalent of a null to a null. It returns *true* in in-memory and DirectQuery models; a provision is made in DirectQuery model to guarantee similar behavior to in-memory model.

Note: in Transact-SQL, a null is never equal to a null. However, in DAX, a blank is equal to another blank. This behavior is the same for all in-memory models. DirectQuery mode uses most SQL Server semantics; but, in this case, it separates from it giving a new behavior to NULL comparisons.

Casts

There is no cast function as such in DAX, but implicit casts are performed in many comparison and arithmetic operations. It is the comparison or arithmetic operation that determines the data type of the result.

For example, Boolean values are treated as numeric in arithmetic operations, such as **TRUE + 1**, or the function `MIN` applied to a column of Boolean values. A `NOT` operation also returns a numeric value.

Boolean values are always treated as logical values in comparisons and when used with **EXACT**, **AND**, **OR**, **&&,** or **||**.

Cast from string to Boolean

In in-memory and DirectQuery models, casts are permitted to Boolean values from these strings only: "" (empty string), "true," "false"; where an empty string casts to false value.

Casts to the Boolean data type of any other string results in an error.

Cast from string to date/time

In DirectQuery mode, casts from string representations of dates and times to actual **datetime** values behave the same way as they do in SQL Server.

Models that use the in-memory data store support a more limited range of text formats for dates than the string formats for dates that are supported by SQL Server. However, DAX supports custom date and time formats.

Cast from string to other non-Boolean values

When casting from strings to non-Boolean values, DirectQuery mode behaves the same as SQL Server. For more information, see `CAST` and `CONVERT` (Transact-SQL).

Cast from numbers to string not allowed

```
EXAMPLE: CONCATENATE(102,”,345”)
```

Casting from numbers to strings is not allowed in SQL Server.

This formula returns an error in tabular models and in DirectQuery mode; however, the formula produces a result in PowerPivot.

No support for two-try casts in DirectQuery

In-memory models often attempt a second cast when the first one fails. This never happens in DirectQuery mode.

```
EXAMPLE: TODAY() + “13:14:15”
```

In this expression, the first parameter has type **datetime** and second parameter has type **string**. However, the casts when combining the operands are handled differently. DAX will perform an implicit cast from string to double. In in-memory models, the formula engine attempts to cast directly to **double**, and, if that fails, it will try to cast the string to **datetime**.

In DirectQuery mode, only the direct cast from string to double will be applied. If this cast fails, the formula will return an error.

Math functions and arithmetic operations

Some mathematical functions will return different results in DirectQuery mode because of differences in the underlying data type or the casts that can be applied in operations. Also, the restrictions described above on the allowed range of values might affect the outcome of arithmetic operations.

Order of addition

When you create a formula that adds a series of numbers, an in-memory model might process the numbers in a different order than a DirectQuery model. Therefore, when you have many large positive numbers and large negative numbers, you may get an error in one operation and results in another operation.

Use of the `POWER` function

```
EXAMPLE: POWER(-64, 1/3)
```

In DirectQuery mode, the `POWER` function cannot use negative values as the base when raised to a fractional exponent. This is the expected behavior in SQL Server.

In an in-memory model, the formula returns *-4*.

Numerical overflow operations

In Transact-SQL, operations that result in a numerical overflow return an overflow error; therefore, formulas that result in an overflow also raise an error in DirectQuery mode.

However, the same formula when used in an in-memory model returns an eight-byte integer. That's because the formula engine doesn't perform checks for numerical overflows.

LOG functions with blanks return different results

SQL Server handles nulls and blanks differently than the xVelocity engine. As a result, the following formula returns an error in DirectQuery mode, but return infinity *(–inf)* in in-memory mode.

```
EXAMPLE: LOG(blank())
```

The same limitations apply to the other logarithmic functions: `LOG10` and `LN`.

Division by 0 and division by Blank

In DirectQuery mode, division by zero (0) or division by BLANK will always result in an error. SQL Server does not support the notion of infinity, and because the natural result of any division by 0 is infinity, the result is an error. However, SQL Server supports division by nulls, and the result must always equal null.

Rather than return different results for these operations, in DirectQuery mode, both types of operations (division by zero and division by null) return an error.

In Excel and in PowerPivot models, division by zero also returns an error. Division by a BLANK returns a BLANK.

The following expressions are all valid in in-memory models, but will fail in DirectQuery mode:

```
1/BLANK
1/0
0.0/BLANK
0/0
```

The expression BLANK/BLANK is a special case that returns BLANK for both in-memory models and DirectQuery mode.

Supported numeric and date-time ranges

Formulas in in-memory tabular model are subject to the same limitations as Excel with regard to maximum allowed values for real numbers and dates. However, differences can arise when the maximum value is returned from a calculation or query, or when values are converted, cast, rounded, or truncated.

If values of types **Currency** and **Real** are multiplied, and the result is larger than the maximum possible value, in DirectQuery mode, no error is raised, and a *null* is returned.

In in-memory models, no error is raised, but the maximum value is returned.

In general, because the accepted date ranges are different for Excel and SQL Server, results can be guaranteed to match only when dates are within the common date range, which is inclusive of the following dates:

Earliest date: March 1, 1990

Latest date: December 31, 9999

If any dates used in formulas fall outside this range, either the formula will result in an error, or the results will not match.

Floating point values supported by `CEILING`

```
EXAMPLE: EVALUATE ROW("x", CEILING(-4.398488E+30, 1))
```

The Transact-SQL equivalent of the DAX `CEILING` function only supports values with magnitude of 10^19 or less. A rule of thumb is that floating point values should be able to fit into **bigint**.

Datepart functions with dates that are out of range

Results in DirectQuery mode are guaranteed to match those in in-memory models only when the date used as the argument is in the valid date range. If these conditions are not satisfied, either an error will be raised, or the formula will return different results in DirectQuery than in in-memory mode.

```
EXAMPLE: MONTH(0) or YEAR(0)
```

In DirectQuery mode, the expressions return *12* and *1899*, respectively.

In in-memory models, the expressions return *1* and *1900*, respectively.

```
EXAMPLE: EOMONTH(0.0001, 1)
```

The results of this expression will match only when the data supplied as a parameter is within the valid date range.

```
EXAMPLE: EOMONTH(blank(), blank()) or EDATE(blank(), blank())
```

The results of this expression should be the same in DirectQuery mode and in-memory mode.

Truncation of time values

```
EXAMPLE: SECOND(1231.04097222222)
```

In DirectQuery mode, the result is truncated, following SQL Server rules, and the expression evaluates to 59.

In in-memory models, the results of each interim operation are rounded; therefore, the expression evaluates to 0.

The following example demonstrates how this value is calculated:

1. The fraction of the input (0.04097222222) is multiplied by 24.
2. The resulting hour value (0.98333333328) is multiplied by 60.
3. The resulting minute value is 58.9999999968.
4. The fraction of the minute value (0.9999999968) is multiplied by 60.
5. The resulting second value (59.999999808) rounds up to 60.
6. 60 is equivalent to 0.

SQL Time data type not supported

In-memory models do not support use of the new SQL **Time** data type. In DirectQuery mode, formulas that reference columns with this data type will return an error. **Time** data columns cannot be imported into an in-memory model.

However, sometimes the engine casts the time value to an acceptable data type, and the formula returns a result.

This behavior affects all functions that use a date column as a parameter.

Currency

In DirectQuery mode, if the result of an arithmetic operation has the type Currency, the value must be within the following range:

Minimum: -922337203685477.5808

Maximum: 922337203685477.5807

Combining currency and real data types

EXAMPLE: Currency sample 1

If **Currency** and **Real** types are multiplied, and the result is larger than 9223372036854774784 (0x7ffffffffffffc00), DirectQuery mode will not raise an error.

In an in-memory model, an error is raised if the absolute value of the result is larger than 922337203685477.4784.

Operation results in an out-of-range value

EXAMPLE: Currency sample 2

If operations on any two currency values result in a value that is outside the specified range, an error is raised in in-memory models, but not in DirectQuery models.

Combining currency with other data types

Division of currency values by values of other numeric types can result in different results.

Aggregation functions

Statistical functions on a table with one row return different results. Aggregation functions over empty tables also behave differently in in-memory models than they do in DirectQuery mode.

Statistical functions over a table with a single row

If the table that is used as argument contains a single row, in DirectQuery mode, statistical functions such as `STDEV` and `VARx` return *null*.

In an in-memory model, a formula that uses `STDEV` or `VARx` over a table with a single row returns a division by zero error.

Text functions

Because relational data stores provide different text data types than Excel, you may see different results when searching strings or working with substrings. The length of strings also can be different.

In general, any string manipulation functions that use fixed-size columns as arguments can have different results.

Additionally, in SQL Server, some text functions support additional arguments that are not provided in Excel. If the formula requires the missing argument you can get different results or errors in the in-memory model.

Operations that return a character using `LEFT`, `RIGHT`, etc. may return the correct character but in a different case, or no results.

Example: `LEFT([“text”], 2)`

In DirectQuery mode, the case of the character that is returned is always exactly the same as the letter that is stored in the database. However, the xVelocity engine uses a different algorithm for compression and indexing of values to improve performance.

By default, the **Latin1_General** collation is used, which is case-insensitive but accent-sensitive. Therefore, if there are multiple instances of a text string in lower case, upper case, or mixed case, all instances are considered the same string, and only the first instance of the string is stored in the index. All text functions that operate on stored strings will retrieve the specified portion of the indexed form. Therefore, the example formula would return the same value for the entire column, using the first instance as the input.

This behavior also applies to other text functions, including `RIGHT`, `MID`, and so forth.

String length affects results

```
EXAMPLE: SEARCH(“within string”, “sample target text”, 1, 1)
```

If you search for a string using the SEARCH function, and the target string is longer than the within string, DirectQuery mode raises an error.

In an in-memory model, the searched string is returned, but with its length truncated to the length of **<within text>.**

```
EXAMPLE: EVALUATE ROW("X", REPLACE("CA", 3, 2, "California") )
```

If the length of the replacement string is greater than the length of the original string, in DirectQuery mode, the formula returns *null*.

In in-memory models, the formula follows the behavior of Excel, which concatenates the source string and the replacement string, which returns *CACalifornia*.

Implicit `TRIM` in the middle of strings

```
EXAMPLE: TRIM(“ A sample sentence with leading white space”)
```

DirectQuery mode translates the DAX `TRIM` function to the SQL statement `LTRIM(RTRIM(<column>))`. As a result, only leading and trailing white space is removed.

In contrast, the same formula in an in-memory model removes spaces within the string, following the behavior of Excel.

Implicit `RTRIM` with use of `LEN` function

```
EXAMPLE: LEN(‘string_column’)
```

Like SQL Server, DirectQuery mode automatically removes white space from the end of string columns: that is, it performs an implicit `RTRIM`. Therefore, formulas that use the `LEN` function can return different values if the string has trailing spaces.

In-memory supports additional parameters for `SUBSTITUTE`

```
EXAMPLE: SUBSTITUTE([Title],”Doctor”,”Dr.”)
EXAMPLE: SUBSTITUTE([Title],”Doctor”,”Dr.”, 2)
```

In DirectQuery mode, you can use only the version of this function that has three (3) parameters: a reference to a column, the old text, and the new text. If you use the second formula, an error is raised.

In in-memory models, you can use an optional fourth parameter to specify the instance number of the string to replace. For example, you can replace only the second instance, etc.

Restrictions on string lengths for `REPT` operations

In in-memory models, the string resulting from an operation using `REPT` must be fewer than 32,767 characters in length.

This limitation does not apply in DirectQuery mode.

Substring operations return different results depending on character type

```
EXAMPLE: MID([col], 2, 5)
```

If the input text is *varchar* or *nvarchar*, the result of the formula is always the same.

However, if the text is a fixed-length character and the value for **<num_chars>** is greater than the length of the target string, in DirectQuery mode, a blank is added at the end of the result string.

In an in-memory model, the result terminates at the last string character, with no padding.