Optimizing DAX Queries

Alberto Ferrari
Senior Consultant
alberto.ferrari@sqlbi.com
www.sqlbi.com
Alberto Ferrari
BI Expert and Consultant
Founder of www.sqlbi.com
Book Writer
Microsoft Business Intelligence Partner
SSAS Maestros – MVP – MCP
Tabular Query Architecture

- **Analysis Services 2012 Tabular Model**
  - Query to **In-Memory Mode**
  - Query to **DirectQuery Mode**

- **In-Memory Mode**
  - Storage Engine Query
  - Read Data from External Sources

- **VertiPaq Storage**
  - SQL Query

- **External Data Sources**
  - Process = Read Data from External Sources

- **DAX / MDX query**
Agenda

• What I will try to teach:
  – Tabular Query Architecture
  – Monitoring and Query Plans
  – How to understand and solve performance issues
  – Optimization Examples

• What I do NOT want you to learn:
  – Best practices, without knowing the reasons
  – Remember: best practices are hints, not rules!
Brief introduction to the SSAS Query Engine Architecture

VERTIPAQ AND FORMULA ENGINE
Tabular Two Engines

• Formula Engine
  – Handles complex expressions
  – Single threaded

• Storage Engine (VertiPaq / xVelocity)
  – Handles simple expressions
  – Executes queries against the database
  – Multithreaded
Tabular: Rich & Fast

DAX
- Rich
- Single threaded per query
- Designed for expressivity

VertiPaq Query
- Simple
- One core per segment
- Optimized for speed
Trust the Rain Man

• Optimizing DAX means:
  – Reduce FE usage
  – Increase SE usage
• It is very easy,
• Only need to understand who processes what
  – What is computed in FE and in SE?
  – How to move computation in SE?
• Time to dive deeper in the DAX engine
Query plans are useful to get insights from the engines in SSAS

USING QUERY PLANS
Understanding Query Plans

- **Logical Query Plan**
  - It is the logical flow of the query
  - Fired as standard text
  - Pretty hard to decode

- **Physical Query Plan**
  - The logical query plan executed by the engine
  - Can be very different from the logical one
  - Uses different operators

- **VertiPaq Queries**
  - Queries executed by the xVelocity engine
DAX Query Optimization Flow

1. Build DAX Expression Tree
2. Build DAX Logical Plan
3. Simplify DAX Logical Plan
4. Build DAX Physical Plan
5. Execute DAX Physical Plan

- Fire Logical Plan Event
- Fire Physical Plan Event

Some Vertipaq queries can be executed in order to gather additional information about data that is to be queried.
DAX is not cost-based

• Unlike SQL Server
• Data is gathered to analyze
  – When to perform filtering
  – How to resolve joins
  – Materialization needs
• Query plan does not change with different row counts
SQL Server Profiler

- Catches events from SSAS
  - Queries Events
  - Query Processing
Monitoring a Query

Trace of a simple query, to start looking at the Profiler

EVALUATE

ROW ( 
    "Result",
    SUM ( 'Internet Sales'[Sales Amount] )
)
Monitoring a Query

Trace of a slightly more complex query, this time we have many VertiPaq SE queries running

EVALUATE

CALCULATETABLE(
  SUMMARIZE(
    'Internet Sales',
    Geography[State Province Code],
    "Sales", SUM ( 'Internet Sales'[Sales Amount] )
  ),
  FILTER(
    Customer,
    Customer[Last Name] = "Anand"
  )
)
The first query is used to understand how many customers exist that satisfy the filter

```
SELECT
    [Customer].[CustomerKey]
FROM
    [Customer]
WHERE
    [Customer].[LastName] = "Anand"
```
2° VertiPaq Query

Query using JOINS inside xVelocity. These joins do not require materialization and can be executed very fast in parallel.

```sql
SELECT
  [Geography].[StateProvinceCode]
FROM [Internet Sales]
  LEFT OUTER JOIN [Customer]
    ON [Internet Sales].[CustomerKey] = [Customer].[CustomerKey]
  LEFT OUTER JOIN [Geography]
    ON [Customer].[GeographyKey] = [Geography].[GeographyKey]
WHERE
  [Customer].[CustomerKey] IN
  (11096, 11989, 17005, 22513, 28899, 15054,
   19626, 20344, 25918, 27141...
  [74 total values, not all displayed]);
```
DAX Query Plan

Simplified text of the query plan, gives a good idea of what the FE is going to execute. In red the parts executed by xVelocity

```
CalculateTable
   AddColumns
     Scan_VertiPaq
     GroupBy_VertiPaq
       Scan_VertiPaq
     Sum_VertiPaq
       Scan_VertiPaq
     'Internet Sales'[Sales Amount]
   Filter_VertiPaq
     Scan_VertiPaq
     'Customer'[Last Name] = Anand
```
Logical plan and Query side by side

Simplified text of the query plan, gives a good idea of what the FE is going to execute. In red the parts executed by xVelocity

```
CalculateTable
  AddColumns
    Scan_VertiPaq
    GroupBy_VertiPaq
      Scan_VertiPaq
    Sum_VertiPaq
      Scan_VertiPaq
        [Sales Amount]
    Filter_VertiPaq
      Scan_VertiPaq
        'Customer'[Last Name] = Anand

Evaluate
  CALCULATETABLE(
    SUMMARIZE(
      'Internet Sales',
      Geography[State Province Code],
      "Sales", SUM( [Sales Amount] )
    ),
    FILTER(
      Customer,
      Customer[Last Name] = "Anand"
    )
  )
```
This query is the first one used to really execute the DAX query. Note that SUM is executed inside xVelocity, not in the Formula Engine.

```
SELECT
    [Geography].[StateProvinceCode],
    SUM([Internet Sales].[SalesAmount])
FROM
    [Internet Sales]
    LEFT OUTER JOIN [Customer]
    ON [Internet Sales].[CustomerKey]=[Customer].[CustomerKey]
    LEFT OUTER JOIN [Geography]
    ON [Customer].[GeographyKey]=[Geography].[GeographyKey]
WHERE
    [Customer].[CustomerKey] IN
    (11096, 11989, ... [74 total values, not all displayed]) VAND
    [Geography].[StateProvinceCode] IN
    ('VIC', 'BC', ... [21 total values, not all displayed]);
```
This query is identical to the second one computed during optimization and will hit the cache

```
SELECT
    [Geography].[StateProvinceCode]
FROM [Internet Sales]
    LEFT OUTER JOIN [Customer]
        ON [Internet Sales].[CustomerKey] = [Customer].[CustomerKey]
    LEFT OUTER JOIN [Geography]
        ON [Customer].[GeographyKey] = [Geography].[GeographyKey]
WHERE
    [Customer].[CustomerKey] IN
    (11096, 11989, 17005, 22513, 28899, 15054,
    19626, 20344, 25918, 27141...
    [74 total values, not all displayed]);
```
Simple Query Plan

• 4 VertiPaq Queries
  – 2 before execution
  – 2 during execution
    • 1 performed GROUPBY and SUM in VertiPaq
    • 1 hit the cache

• Intermediate results are cached
• Final JOIN performed by Formula Engine
• This is a very good query plan
Other useful info in the Profiler

• There are other useful info
  – CPU Time
  – Duration
  – Event Subclass

• CPU Time >= Duration
  – Many cores run in parallel
  – Only during VertiPaq scans
  – Formula Engine is still single-threaded
Understanding cache usage is one of the key of optimization

THE VERTIPAQ CACHE
What is cached?

- Results of VertiPaq SE Queries are cached
- Nothing else
- Thus, result of FE calculation is not cached
- For MDX queries, the MDX cache is available too, resulting in better cache options
Clear the Cache

Always remember to clear the cache prior to execute any performance test, otherwise numbers will be contaminated. That said, check cache usage too, when optimizing a measure. It is important!

```xml
  <ClearCache>
    <Object>
      <DatabaseID>Adventure Works DW Tabular</DatabaseID>
    </Object>
  </ClearCache>
</Batch>
```
Does SUMX Really Iterate?

This SUMX is resolved inside VertiPaq because it is a simple operation that VertiPaq know how to handle. No iteration happens here.

EVALUATE
ROW (
  "Sum",
  SUMX(
    'Internet Sales',
    'Internet Sales'[Sales Amount] / 'Internet Sales'[Order Quantity] )
)

SELECT
  SUM( [Internet Sales].[SalesAmount] / [Internet Sales].[OrderQuantity] )
FROM
  [Internet Sales];
A Story of CallBackDataID

If the expression is too complex, CallBackDataID appears, meaning a call back to formula engine during the VertiPaq scan.

```sql
EVALUATE
ROW (
    "Sum",
    SUMX ( 'Internet Sales',
        IF (
            'Internet Sales'[Sales Amount] > 0,
            'Internet Sales'[Sales Amount] / 'Internet Sales'[Order Quantity]
        )
    )
);
```

```sql
SELECT SUM(
    [CallbackDataID( IF (
        'Internet Sales'[Sales Amount] > 0,
        'Internet Sales'[Sales Amount] / 'Internet Sales'[Order Quantity])]
))
FROM [Internet Sales];
```
CallBackDataID in Action

VERTIPAQ SCAN

<table>
<thead>
<tr>
<th>SalesAmount</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,500</td>
<td>12</td>
</tr>
<tr>
<td>3,500</td>
<td>14</td>
</tr>
<tr>
<td>12,500</td>
<td>12</td>
</tr>
</tbody>
</table>

FORMULA ENGINE

IF (Sales Amount > 0, Sales Amount / Quantity)

Aggregation computed inside Vertipaq, no spooling was necessary

SELECT SUM([CallbackDataID](
    IF ('Internet Sales'[Sales Amount] > 0,
        'Internet Sales'[Sales Amount] / 'Internet Sales'[Order Quantity]
    )
))
FROM [Internet Sales];
CallBackDataID Performance

• Slower than pure Vertipaq
• Faster than pure Formula Engine
  – Highly parallelized
  – Works on compressed data
• Not cached
The Test Data Model

- Networks
  - PK
  - ID_Network
  - Network

- Audience
  - PK, FK1
  - ID_Individual
  - ID_Date
  - ID_Time
  - ID_Network

- Calendar
  - PK
  - ID_Date
  - WeekDay

- Time
  - PK
  - ID_Time
  - Hour
  - Minute

- Individuals
  - PK
  - ID_Individual
  - Individual

- TargetsForIndividuals
  - PK, FK1
  - ID_Target
  - ID_Individual
  - DateStart
  - DateEnd

- Targets
  - PK
  - ID_Target
  - Target

Numbers:
- 9,000
- 176,000
- 2,705,081
- 4,058,451,434
- 1,440
- 230
- 140
Why SUMMARIZE is slower than ADDCOLUMNS?
Many ways to SUMMARIZE

• Best practice:
  – Use ADDCOLUMNS instead of SUMMARIZE

• Why?
  – Let us use the profiler to understand it
  – And get a sense out of the statement
Some other best practices...

OTHER OPTIMIZATION TECHNIQUES
Currency Conversion

Two SUMX are converted to two iterations

\[
[\text{FirstCurrencyAmount}] := \text{SUMX}(\text{DimCurrency}, \text{SUMX}(\text{DimDate}, \text{CALCULATE}(\text{VALUES}(\text{CurrencyRate}[\text{AverageRate}]) * \text{SUM}(\text{FactInternetSales}[\text{SalesAmount}]))))
\]
Currency Conversion

Simple SUMX over CrossJoin is resolved as a single VertiPaq scan with CallbackDataID to compute the multiplication.

\[
\text{[SecondCurrencyAmount]} := \text{SUMX(}
\text{CROSSJOIN(}
\text{DimCurrency, DimDate}
\text{),}
\text{CALCULATE(}
\text{VALUES( CurrencyRate[AverageRate] )}
\text{* SUM( FactInternetSales[SalesAmount] )}
\text{)}
\text{)}
\]
This query computes YTD and QTD inside a loop and then removes empty values.

```
DEFINE
  MEASURE 'Internet Sales'[Sales] =
    CALCULATE( ROUND( SUM( 'Internet Sales'[Sales Amount] ), 0 ) )
  MEASURE 'Internet Sales'[YTD Sales] =
    TOTALYTD( [Sales] , 'Date'[Date] )
  MEASURE 'Internet Sales'[QTD Sales] =
    TOTALQTD( [Sales] , 'Date'[Date] )
EVALUATE
FILTER(
  ADDCOLUMNS(
    CROSSJOIN(
      VALUES( 'Date'[Calendar Year] ),
      VALUES( 'Date'[Month] ),
      VALUES( 'Date'[Month Name] )
    ),
    "Sales", [Sales],
    "YTD Sales", [YTD Sales],
    "QTD Sales", [QTD Sales]
  ),
  NOT ISBLANK( [Sales] )
)
ORDER BY 'Date'[Calendar Year], 'Date'[Month]
```
Filter as Soon as You Can

This second query removes empty rows before computing expensive measures. It runs 7 times faster even if it computes [Sales] twice.

```plaintext
DEFINE
  MEASURE 'Internet Sales'[Sales] =
    CALCULATE( ROUND( SUM( 'Internet Sales'[Sales Amount] ), 0 ) )
  MEASURE 'Internet Sales'[YTD Sales] =
    TOTALYTD( [Sales] , 'Date'[Date] )
  MEASURE 'Internet Sales'[QTD Sales] =
    TOTALQTD( [Sales] , 'Date'[Date] )
EVALUATE
ADDCOLUMNS(
  FILTER(
    CROSSJOIN(
      VALUES( 'Date'[Calendar Year] ),
      VALUES( 'Date'[Month] ),
      VALUES( 'Date'[Month Name] )
    ),
    NOT ISBLANK( [Sales] )
  ),
  "Sales", [Sales],
  "YTD Sales", [YTD Sales],
  "QTD Sales", [QTD Sales]
)
ORDER BY 'Date'[Calendar Year], 'Date'[Month]
```
Conclusions

• DAX is not easy to optimize
  – Otherwise, it would be boring...
• Query plans make it possible to optimize
• Learn the difference between FE and SE
• Test, test, test!
• Never trust simple rules, including mine 😊
THANK YOU!

• For attending this session and PASS SQLRally Nordic 2013, Stockholm