Microsoft

U-SQL - Performance optimization

Performance optimization

Prepared by:
Arshad Ali
Data Insights COE Associate Architect

Technical reviewers:
Andy Isley
Manav Gupta

Note: The detail provided in this document has been harvested as part of a customer engagement sponsored through the Azure Data Services Jumpstart Program.

Table of contents

Introduction

This document provides details of some points to consider for performance optimization of U-SQL tables and the queries running against them.

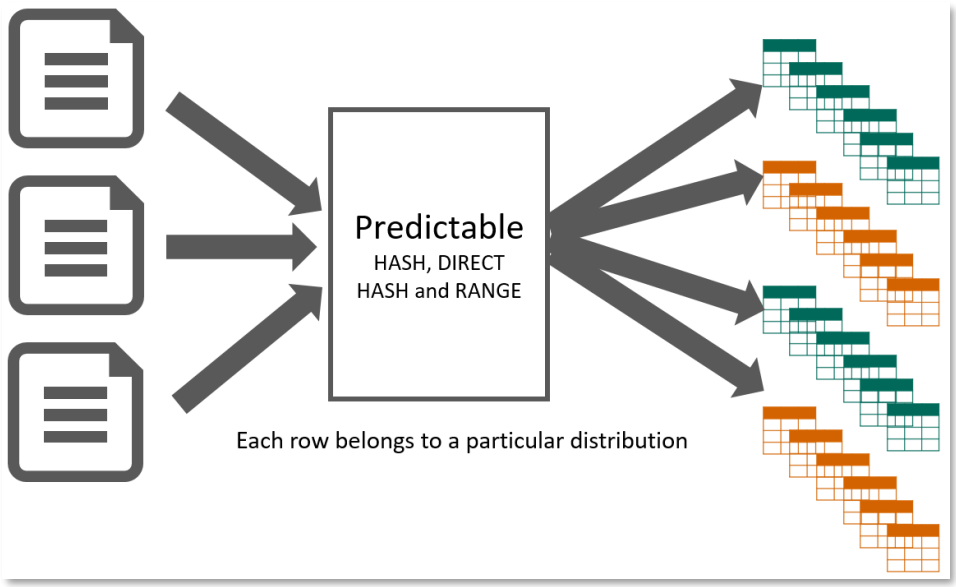
Performance optimization

Predictable vs. un-predictable distribution

The real power of a distributed system and distributed processing comes from the fact data is processed in distributed manners by multiple computers and by minimizing data movement across these computers during data processing. Though the U-SQL engine has been designed to process the data where it exists (moving the compute to the data rather than moving data to the compute), it also provides the designer with some control over deciding the data distribution strategy for a U-SQL table. U-SQL provides four different distributions (HASH, RANGE, ROUND ROBIN, and DIRECT HASH). These four distribution schemes can be broadly groups into two categories: *predictable distribution* and *un-predictable distribution*.

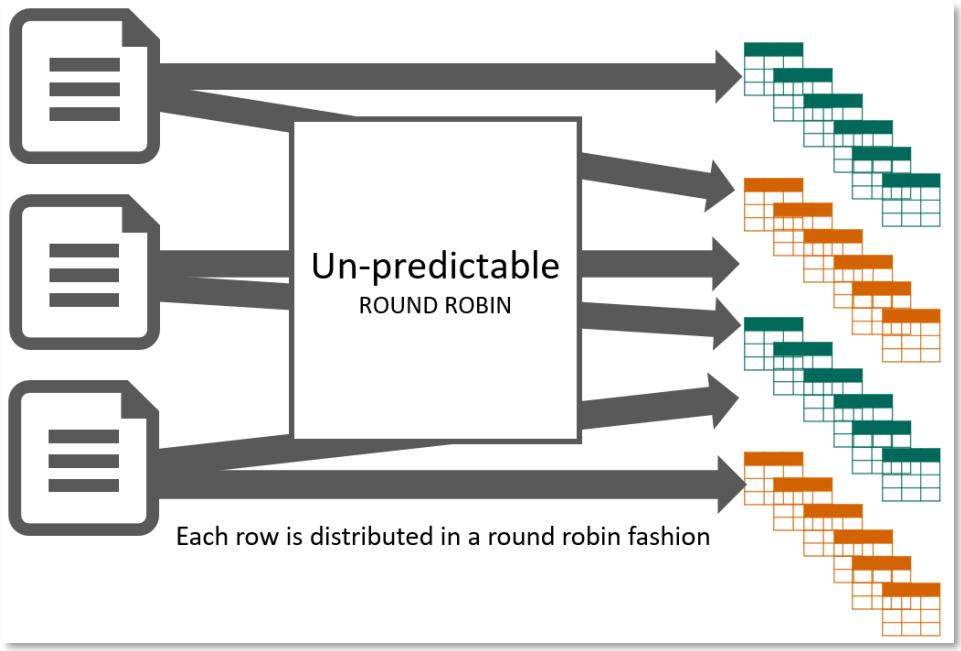
Predictable distribution

HASH, DIRECT HASH, and RANGE distribution schemes provide predictable behavior by placing a row to a specific distribution bucket based on key(s) provided. The selection of the distribution key is done by considering certain factors such as minimizing the data skewness, minimizing the data shuffling during query execution, and the types of queries being executed on the system.



Un-predictable distribution

When you use a ROUND ROBIN distribution scheme, the data is evenly (or as evenly as possible) distributed among all the distribution buckets randomly. Round-robin tables do not exhibit signs of data skewness because the data is stored evenly across the distribution buckets. However, query execution might involve lots of data shuffling (based on the size of the data and the query being executed) among distribution buckets.

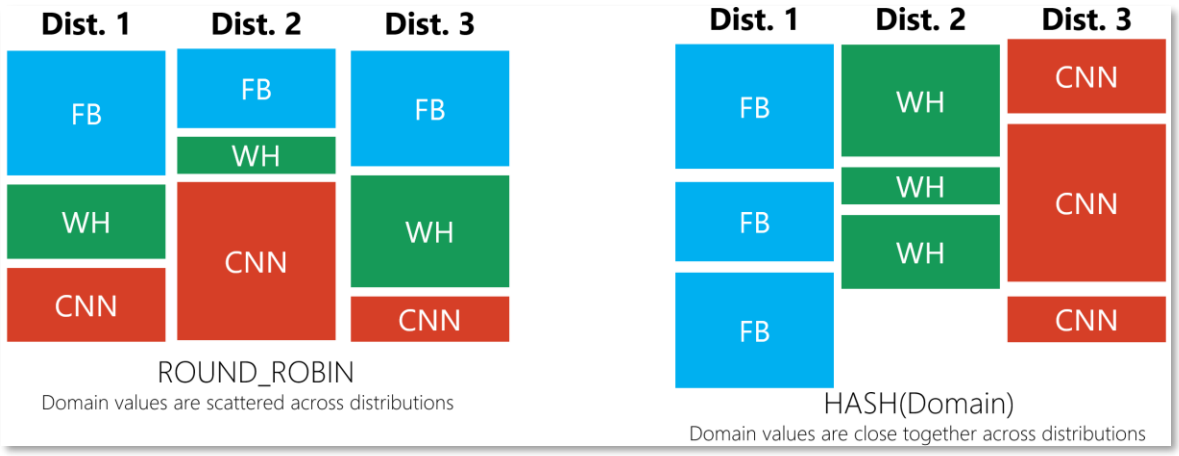


Hashing vs. round robin

Now let's deep dive into the performance impact of selecting various types of distribution schemes. For example, consider the following dataset stored in into two tables, one that uses ROUND ROBIN as the distribution scheme and the other, which is hashed on Domain column:

SessionID	Domain	Clicks
3	cnn.com	9
1	whitehouse.gov	14
2	facebook.com	8
3	reddit.com	78
2	microsoft.com	1
1	facebook.com	5
3	microsoft.com	11

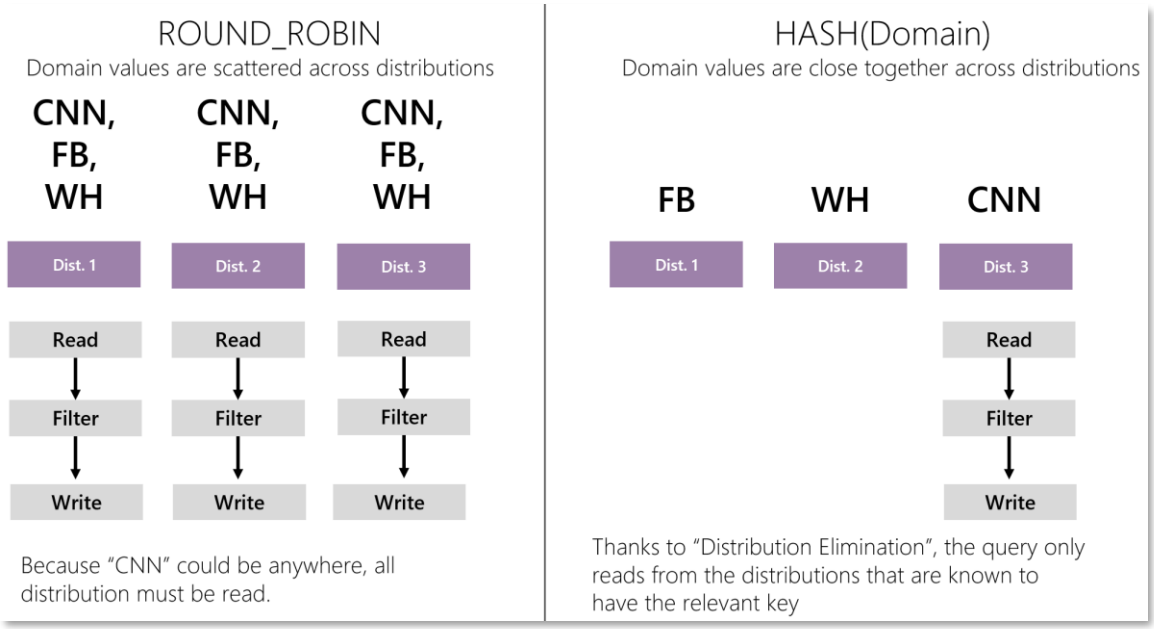
When you store the data in the two tables above, the data distribution for both tables should be similar to that shown below (for simplicity, I have not shown all the data and distributions):



Now, execute the query below to determine the number of clicks for the CNN domain only, and then examine the execution plan:

```
@ClickData =
    SELECT *
    FROM MyDB.dbo.<<TableName>>;

@rows =
    SELECT *
    FROM @ClickData
    WHERE Domain == "cnn.com";

OUTPUT @rows
    TO "/output.tsv"
    USING Outputters.tsv();
```

As expected, a query on a ROUND ROBIN table scans through all distributions (as CNN could be in any of those distributions), whereas in the case of a query on the other table (Hashed on Domain), the U-SQL engine knows the specific distributions (which are known to have "cnn.com" value) to scan and eliminates the other distributions during processing. That's how your table design impacts the performance, minimizes the data movement\processing time, and uses the true power or a distributed system. It's equally important to choose the right distribution hash key based on your query patterns (joins, filters, and groupings) and data distribution (to avoid data stickiness).

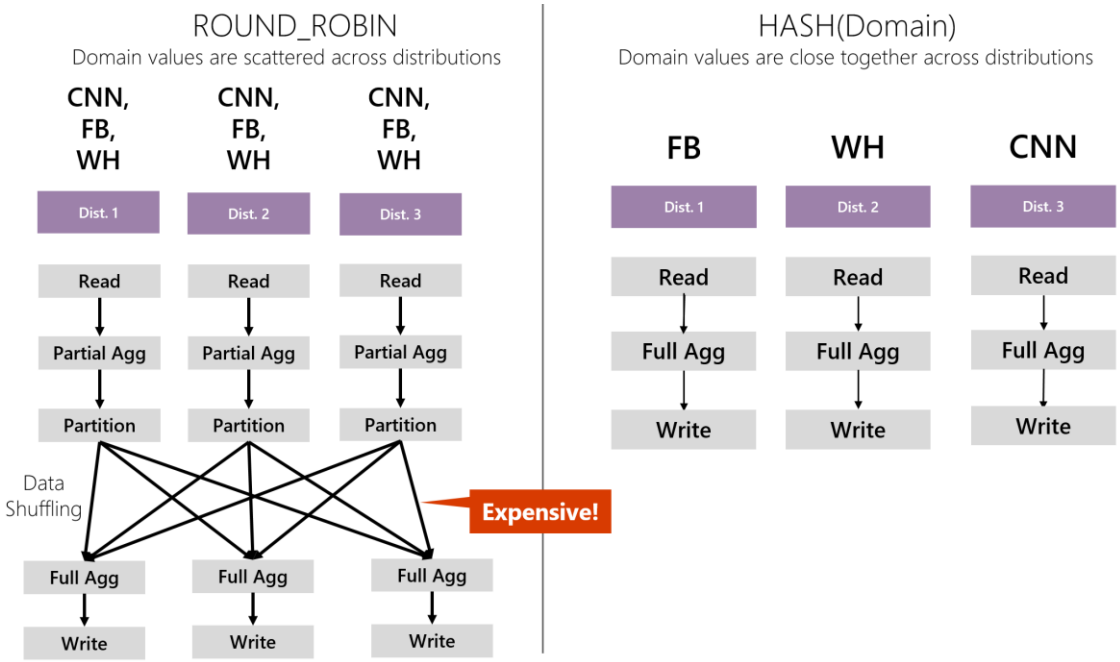


Now, execute below query to get number of clicks for each domain and examine the execution plan:

```
@ClickData =
    SELECT *
    FROM MyDB.dbo.<<TableName>>;

@rows =
    SELECT Domain,
           SUM(Clicks) AS TotalClicks
```

```
    FROM @ClickData
    GROUP BY Domain;

OUTPUT @rows
    TO "/output.tsv"
    USING Outputters.tsv();
```

As shown in the figure below, the dataset and distribution elimination does not apply in the HASH(Domain), but with the ROUND_ROBIN table there is an overhead of data shuffling during the aggregation process across distributions. Each domain can be in either of the distributions, but in the case of a hashed table, the U-SQL engine does the full aggregation.



Benefits of distribution with hashing

As previously discussed in detail, the hashing distribution scheme leverages the power of a distributed system by processing data with minimal data scan and data shuffling. A summary of the benefits and considerations when choosing the best distribution hashing key follows:

- Design for the most frequent/costly queries.
- Manage data skew in partition/table (the distribution key should contain significant distinct values, so it's better to avoid the use of a nullable column as it could be a bad candidate, especially if you have substantial number of null values).
- Manage parallelism in querying (by number of distributions).
- Manage minimizing data movement in joins and groupings.
- Ensure that distribution keys are a prefix of Clustered Index keys, to:
    - Manage data skew in the distribution bucket.
    - Provide locality of the same data values.
    - Provide seeks and range scans for query predicates (index lookup).
- Provide distribution seeks and range scans for query predicates (distribution elimination).

Optimization with partitioning

Partitioning helps in loading incremental daily or hourly data, by way of deleting the existing partition (containing old data) and adding a new partition (with new data). But even more importantly, partitioning plays role in performance optimization by techniques called partition elimination and parallel processing. Let's demonstrate how these techniques work in U-SQL query execution and how the U-SQL engine optimizes performance.

First, let's create two distributed tables (one partitioned and other one non-partitioned) and load 12 months' data into them.

```
DROP TABLE IF EXISTS FactInternetSales_NonPartitioned;
CREATE TABLE FactInternetSales_NonPartitioned
(
    ProductKey int,
    OrderDateKey int,
    DueDateKey int,
    ShipDateKey int,
    CustomerKey int,
    PromotionKey int,
    CurrencyKey int,
    SalesTerritoryKey int,
    SalesOrderNumber string,
    SalesOrderLineNumber int,
    RevisionNumber int,
    OrderQuantity int,
    UnitPrice decimal,
    ExtendedAmount decimal,
    UnitPriceDiscountPct decimal,
    DiscountAmount decimal,
    ProductStandardCost decimal,
    TotalProductCost decimal,
    SalesAmount decimal,
    TaxAmt decimal,
    Freight decimal,
    CarrierTrackingNumber string,
    CustomerPONumber string,
    OrderDate DateTime,
    DueDate DateTime,
    ShipDate DateTime,
    CalendarYearMonth string,
    INDEX IDX_FactInternetSales_NonPartitioned_CustomerKey CLUSTERED(CustomerKey ASC)
    DISTRIBUTED BY HASH(CustomerKey) INTO 25
);
@HistoricalData =
    EXTRACT ProductKey int,
            OrderDateKey int,
            DueDateKey int,
            ShipDateKey int,
            CustomerKey int,
            PromotionKey int,
            CurrencyKey int,
            SalesTerritoryKey int,
```

```
            SalesOrderNumber string,
            SalesOrderLineNumber int,
            RevisionNumber int,
            OrderQuantity int,
            UnitPrice decimal,
            ExtendedAmount decimal,
            UnitPriceDiscountPct decimal,
            DiscountAmount decimal,
            ProductStandardCost decimal,
            TotalProductCost decimal,
            SalesAmount decimal,
            TaxAmt decimal,
            Freight decimal,
            CarrierTrackingNumber string,
            CustomerPONumber string,
            OrderDate DateTime,
            DueDate DateTime,
            ShipDate DateTime,
            CalendarYearMonth string
    FROM "/Data/Fact/FactInternetSales_2013.csv"
    USING Extractors.Csv(skipFirstNRows : 1);

INSERT FactInternetSales_NonPartitioned
SELECT * FROM @HistoricalData;
```

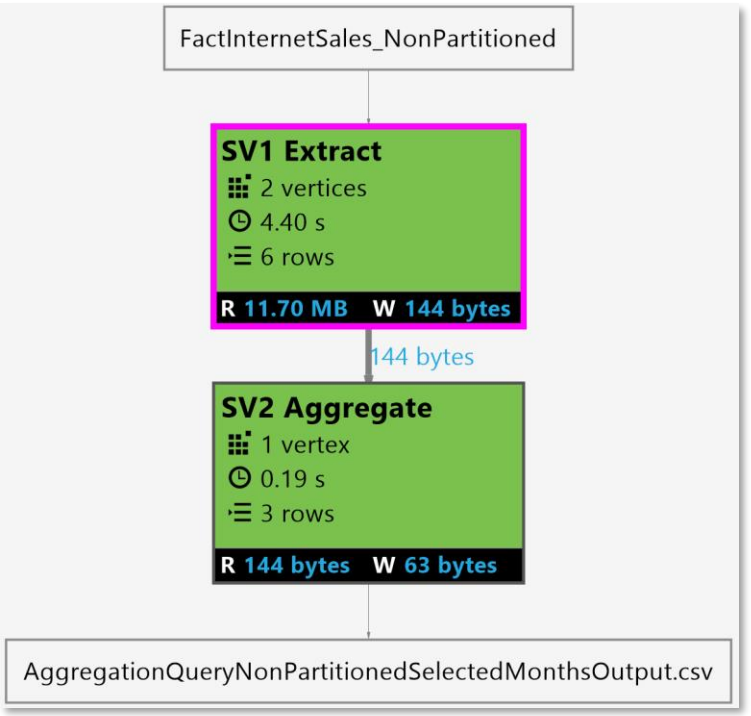
```
DROP TABLE IF EXISTS FactInternetSales_Partitioned;
CREATE TABLE FactInternetSales_Partitioned
(
    ProductKey int,
    OrderDateKey int,
    DueDateKey int,
    ShipDateKey int,
    CustomerKey int,
    PromotionKey int,
    CurrencyKey int,
    SalesTerritoryKey int,
    SalesOrderNumber string,
    SalesOrderLineNumber int,
    RevisionNumber int,
    OrderQuantity int,
    UnitPrice decimal,
    ExtendedAmount decimal,
    UnitPriceDiscountPct decimal,
    DiscountAmount decimal,
    ProductStandardCost decimal,
    TotalProductCost decimal,
    SalesAmount decimal,
    TaxAmt decimal,
    Freight decimal,
    CarrierTrackingNumber string,
    CustomerPONumber string,
    OrderDate DateTime,
    DueDate DateTime,
```

```
    ShipDate DateTime,
    CalendarYearMonth string,
    INDEX IDX_FactInternetSales_Partitioned_CustomerKey CLUSTERED(CustomerKey ASC)
    PARTITIONED BY (CalendarYearMonth)
    DISTRIBUTED BY HASH(CustomerKey) INTO 25
);
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201301"); //Jan, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201302"); //Feb, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201303"); //Mar, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201304"); //Apr, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201305"); //May, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201306"); //Jun, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201307"); //Jul, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201308"); //Aug, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201309"); //Sep, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201310"); //Oct, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201311"); //Nov, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("201312"); //Dec, 2013
ALTER TABLE FactInternetSales_Partitioned ADD PARTITION ("209912"); //Default Partition
@HistoricalData =
    EXTRACT ProductKey int,
            OrderDateKey int,
            DueDateKey int,
            ShipDateKey int,
            CustomerKey int,
            PromotionKey int,
            CurrencyKey int,
            SalesTerritoryKey int,
            SalesOrderNumber string,
            SalesOrderLineNumber int,
            RevisionNumber int,
            OrderQuantity int,
            UnitPrice decimal,
            ExtendedAmount decimal,
            UnitPriceDiscountPct decimal,
            DiscountAmount decimal,
            ProductStandardCost decimal,
            TotalProductCost decimal,
            SalesAmount decimal,
            TaxAmt decimal,
            Freight decimal,
            CarrierTrackingNumber string,
            CustomerPONumber string,
            OrderDate DateTime,
            DueDate DateTime,
            ShipDate DateTime,
            CalendarYearMonth string
    FROM "/Data/Fact/FactInternetSales_2013.csv"
    USING Extractors.Csv(skipFirstNRows : 1);

INSERT FactInternetSales_Partitioned ON INTEGRITY VIOLATION MOVE TO PARTITION
("209912")
SELECT * FROM @HistoricalData;
```

Now let's run the following query to summarize the sales amount for each month and analyze the execution plan for the non-partitioned table:

```
//Querying data for all the months
@AggregationQueryNonPartitionedAllMonths =
    SELECT CalendarYearMonth, SUM(SalesAmount) AS TotalSalesAmount
    FROM FactInternetSales_NonPartitioned
    GROUP BY CalendarYearMonth;

OUTPUT @AggregationQueryNonPartitionedAllMonths
    TO "/Data/Fact/Output/AggregationQueryNonPartitionedAllMonthsOutput.csv"
USING Outputters.Csv();
```

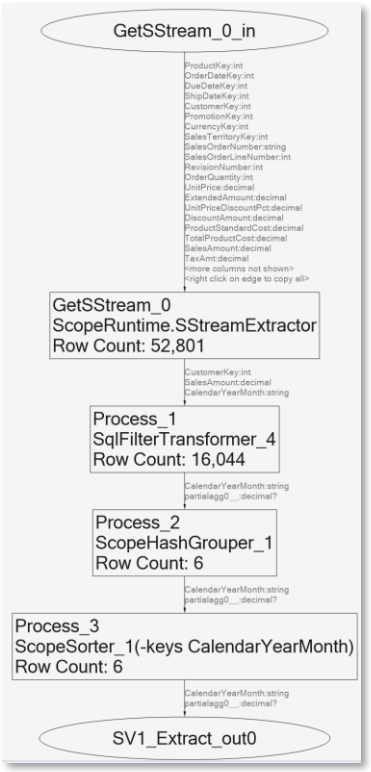
As shown in the figure below, the query reads all the data from that table (~12 MB) and then it does summarization by grouping on month.



Now let's execute the following script to summarize the sales amount for the last 3 months only. Again, the query reads all the data from that table (~12 MB), applies a filter to consider only the last three months of data, and then does summarization by grouping on those selected months.

```
//Querying data for selected months only
@AggregationQueryNonPartitionedSelectedMonths =
    SELECT CalendarYearMonth,
           SUM(SalesAmount) AS TotalSalesAmount
    FROM FactInternetSales_NonPartitioned
    WHERE CalendarYearMonth IN ("201312", "201311", "201310")
    GROUP BY CalendarYearMonth;

OUTPUT @AggregationQueryNonPartitionedSelectedMonths
    TO "/Data/Fact/Output/AggregationQueryNonPartitionedSelectedMonthsOutput.csv"
USING Outputters.Csv();
```

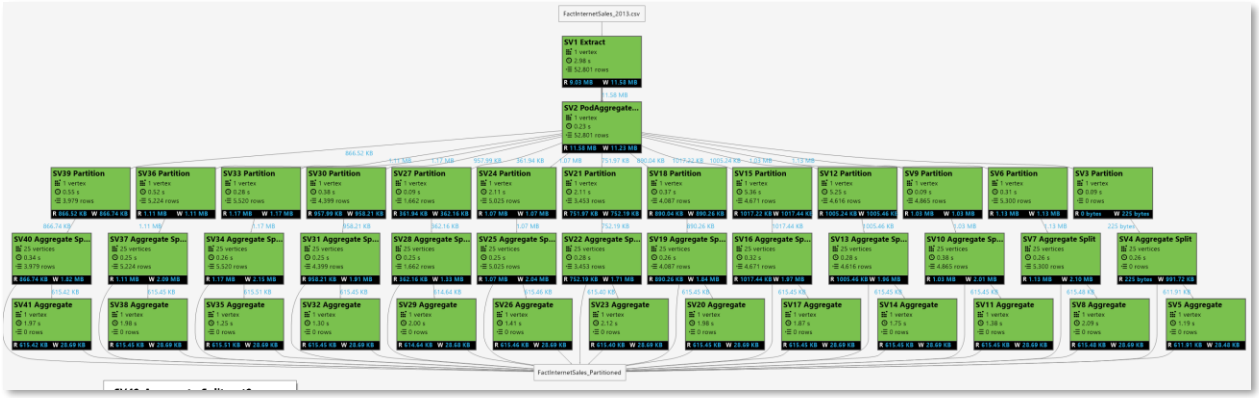
You can verify this by expanding the vertex, which indicates that the query reads 52801 rows (total rows in the table) and applies a filter to limit processing to the last 3 months of data (16044 rows) only.



Now let's run through the same exercise using the partitioned table:

```
//Querying data for all the months
@AggregationQueryPartitionedAllMonths =
    SELECT CalendarYearMonth, SUM(SalesAmount) AS TotalSalesAmount
    FROM FactInternetSales_Partitioned
    GROUP BY CalendarYearMonth;

OUTPUT @AggregationQueryPartitionedAllMonths
    TO "/Data/Fact/Output/AggregationQueryPartitionedAllMonthsOutput.csv"
USING Outputters.Csv();
```

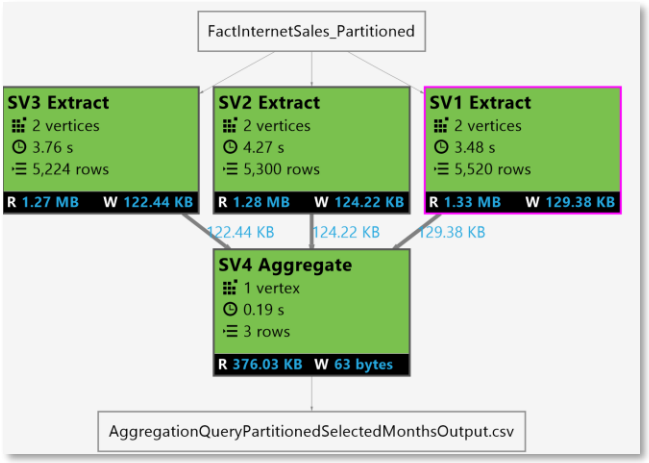
Notice that the query again reads all the data from the table as we want summarization of sales amount by month for all the available months and we haven't specified any filter condition. Notice this time that there are 13 vertices (one for each month [12] plus 1 default Catch-All partition) working together in parallel. This is a very small dataset and hence you will not notice much difference in the execution time. However, consider processing a larger data set and the amount of time that you can save by leveraging this parallel execution (after all, U-SQL is designed and optimized for big data processing).



Now let's execute the following script to summarize the sales amount for the last 3 months only from the partitioned table. In this case, notice that the query only reads data from partitions for last 3 months.

```
//Querying data for selected months only
@AggregationQueryPartitionedSelectedMonths =
    SELECT CalendarYearMonth,
           SUM(SalesAmount) AS TotalSalesAmount
    FROM FactInternetSales_Partitioned
    WHERE CalendarYearMonth IN ("201312", "201311", "201310")
    GROUP BY CalendarYearMonth;

OUTPUT @AggregationQueryPartitionedSelectedMonths
    TO "/Data/Fact/Output/AggregationQueryPartitionedSelectedMonthsOutput.csv"
USING Outputters.Csv();
```

Behind the scenes, the U-SQL engine, eliminates all the data or partitions that are not needed for your query execution and only considers the last 3 months' partitions (~4MB or 16044 rows) and processes all these partitions in parallel.



Appendix

Data files used in example

Datafiles.zip

Solution (USQL and .NET projects)

LearnUSQL.zip

Reference

- U-SQL Language Reference
- Tutorial: Get started U-SQL
- U-SQL Tables
- U-SQL Partitioned Data and Tables
- U-SQL Query Execution and Performance Tuning

Feedback and suggestions

If you have feedback or suggestions for improving this data migration asset, please contact the Data Migration Jumpstart Team (askdmjfordmtools@microsoft.com). Thanks for your support!

Note: For additional information about migrating various source databases to Azure, see the Azure Database Migration Guide.