Microsoft

# Azure Data Lake Analytics – U-SQL

## Table design, incremental data load, and partition management

*Prepared by:*
**Arshad Ali**
Data Insights COE Associate Architect

**Andy Isley**
Data Insights COE Architect

*Technical reviewer:*
**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

Azure Data Lake Analytics (U-SQL) originates from the world of Big Data, in which data is processed in a scale-out manner by using multiple nodes. These nodes can access the data in several formats, from flat files to U-SQL tables.

This document focuses on U-SQL tables, with details about table design strategy, different types of partitions that can be created on U-SQL tables and how they differ from each other, the process for incrementally loading data into a partitioned table, and how to manage partitions.

# Partition management

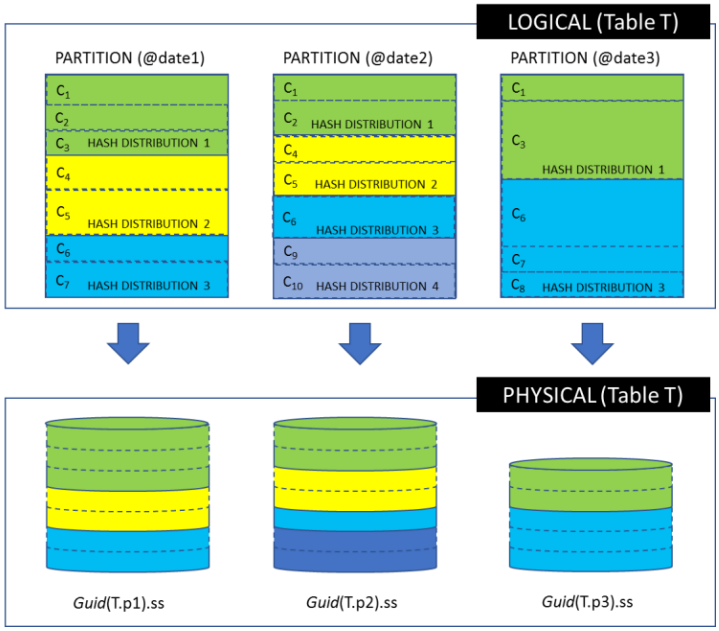
A primary consideration when working with Azure Data Lake Analytics (ADLA) is how the data can be stored. ADLA supports two types of tables (Schema-On-Read and Schema-On-Write) as well as two types of data partitioning (Distributions and Partitions), which enable parallel processing and performance optimizations.

## Types of tables

As with other SQL-inspired Big Data processing systems such as Hive, U-SQL provides an option to create either an external table or a managed table that contains schematized structured data. This whitepaper primarily focuses on managed tables, as they provide additional optimizations such as partitioning and distribution, while external tables do not.

When a table is partitioned, the logical table is split into several individually addressable partition buckets (physical files). These partition buckets can further be split into distribution buckets by using a distribution key (Hash/Direct Hash(id)/Range/Round Robin). The purpose of distributions is to group similar data together for faster access.

```
TABLE T ( id …, C …, date DateTime, …
   , INDEX i CLUSTERED (id, C)
     PARTITIONED BY (date)
     DISTRIBUTED BY HASH (id) INTO 4)
```



# Partitioning and distributions

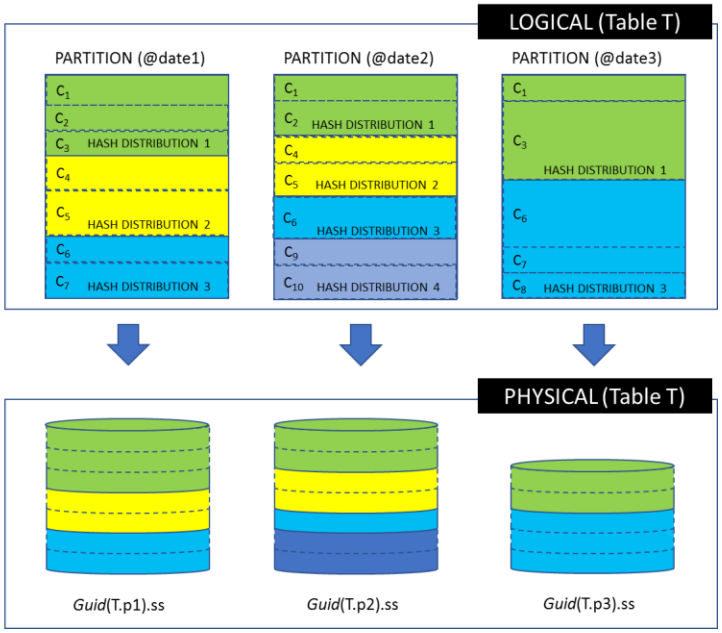
## Partitioning (a.k.a. Course-Grained or Vertical Partitioning)

Data life cycle management is one reason to use partitions. Creating partitions allows Data Lake Analytics to address a specific partition for processing. Often, data is loaded into Data Lake based on daily or hourly data feeds. These feeds can select a specific partition and Add/Update the data within the partition without having to work with all the data. To accomplish this, each partition must be explicitly added with ALTER TABLE ADD PARTITION and removed with ALTER TABLE DROP PARTITION.

An additional benefit is that the query processor can access each partition in parallel and will perform partition elimination. This greatly benefits the query engine, as partitions that are outside of the predicate where clause are not required to fulfill the result set.

Think of each of these partitions as smaller sub tables with the same schema, which contains rows matching with value in date column, as shown in the following graphic.

```
TABLE T ( id …, C …, date DateTime, …
   , INDEX i CLUSTERED (id, C)
     PARTITIONED BY (date)
     DISTRIBUTED BY HASH (id) INTO 4)
```




The script below is designed to create a table with vertical partitioning definition on the CalendarYearMonth column.

```
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
);
```

An additional step is needed to create the individual partitions, by using an Alter Table command.

If you wanted to load data by month for 2013, you could create 12 partitions, one partition for each month. You would also create a default partition that is used as a catch all partition. This is considered a best practice when working with partitions in Azure Data Lake (ADL). The below script creates a partition for each month of 2013, along with a default partition. The default/Catch-All partition is used to store records that don't get placed into any of the other partitions because their values are beyond the current partitions' boundaries.

```
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
- Empty
```

## Distributions (a.k.a. Fine-grained Distributions or Horizontal Partitioning)

U-SQL supports distributing a table across each of the partitions created in a managed table. An unpartitioned table will contain a single partition (Default). Distributions further break down a partition into smaller segments. Creating distributions is accomplished by using the Distribution clause in the create statement. Distributions group similar data.

The way a table is partitioned depends on the data and the types of queries being run against the data. Partitions should be used for:

- Design for most frequent/costly queries
- Minimize data movement at query time.
- Mange data skew in partitions.
- Parallel data processing in ADLA.
- Distribution elimination.

Currently, U-SQL supports four distribution schemes: HASH, RANGE, ROUND ROBIN, and DIRECT HASH, as described in the following table

| Distribution scheme | Description |
|---|---|
| HASH | Tables distributed by HASH requires a list of columns (one or more) that will be used to route each row individually to a distribution bucket based on a hash of the columns specified. Like data will always generate the same hash and therefore be located within the same distribution. |
| DIRECT HASH | Uses a single integer column as the distribution bucket. This is often used with ranking functions such as RANK() and DENSE_RANK(). |
| RANGE | This works in way similar to HASH – you provide the key (or ordered keys), and the data will be distributed across the buckets. |
| ROUND ROBIN | This is used when a distribution key can't be found in the data or when all other distribution options cause data skew within the distributions. This will evenly distribute the data, however, similar data will not be grouped together. This is a good "Catch-All" model to use when none of the other schemes meet your requirements. |

The create a table statement below uses a hash distributed on column "CustomerKey" to slice the data into 25 individual distributions, or distribution buckets.

```
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
);
```

The INTO clause in the above script specifies the number of buckets into which the data will be distributed. The number of distributions must be between 2 and 2500. If INTO is not specified, the data will be distributed across 25 distributions.

Remember, if you have a table with 10 partitions, the table is distributed into 25 distributions buckets. It means total number of distribution buckets are equal to total number of partitions multiplied by defined distribution buckets (10 * 25 = 250 distributions). Keep this formula in mind, as distributions with only a few records can impact performance negatively.

# Incremental data load

## Handling INSERT only

Inserting new dataset to a U-SQL table is simple and straight forward. In the script below, you first create a rowset variable with the EXTRACT statement pointing to the file (or files), and then use the INSERT INTO statement to load the data into the destination U-SQL table.

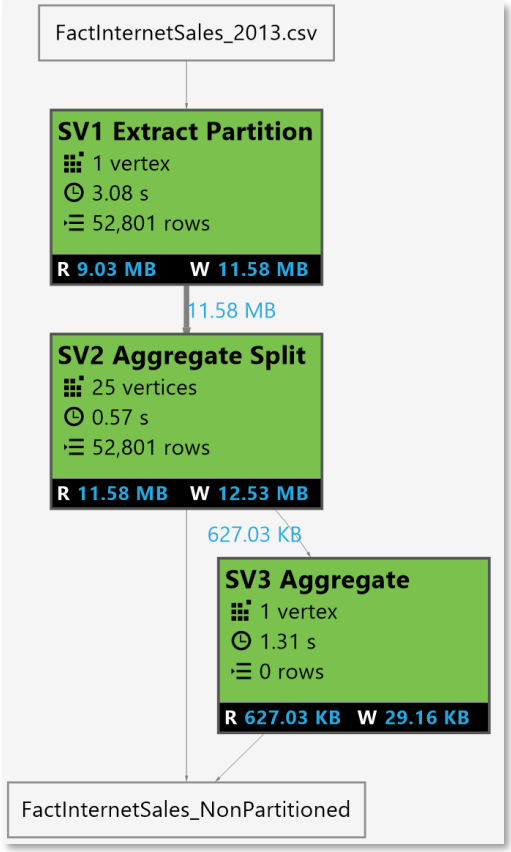
```
DROP TABLE IF EXISTS FactInternetSales_NonPartitioned;
CREATE TABLE FactInternetSales_NonPartitioned
(
    ProductKey int,
    OrderDateKey int,
    DueDateKey int,
    ShipDateKey int,
    CustomerKey int,
    PromotionKey int,
    CurrencyKey int,
    SalesTerritoryKey int,
    SalesOrderNumber 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;
```

The above script creates a U-SQL distributed table and loads data into it with the INSERT statement.

The figure below shows an execution plan for the previous statement. The data is loaded into a U-SQL table with 25 distribution buckets (the Default number of distributions). This script reads data from the file (1 vertex in this case as file size is very small; if the file size is more than 1 GB it creates one vertex for each 1 GB and then run them in parallel) and then it loads data into the destination U-SQL tables (25 vertices, one vertex to load into each distributed bucket).




If we change the table definition to have Partitions by CalYearMonth, the execution plan is quite different. The script below adds Partitions to the destination table. It also includes an ON INTEGRITY VIOLATION statement to make sure that all incoming records are placed into the destination table. Use this clause to ensure that there is an available partition for all the new data. If this statement were omitted, the data will be either ignored and not loaded into the table.

```
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;
```

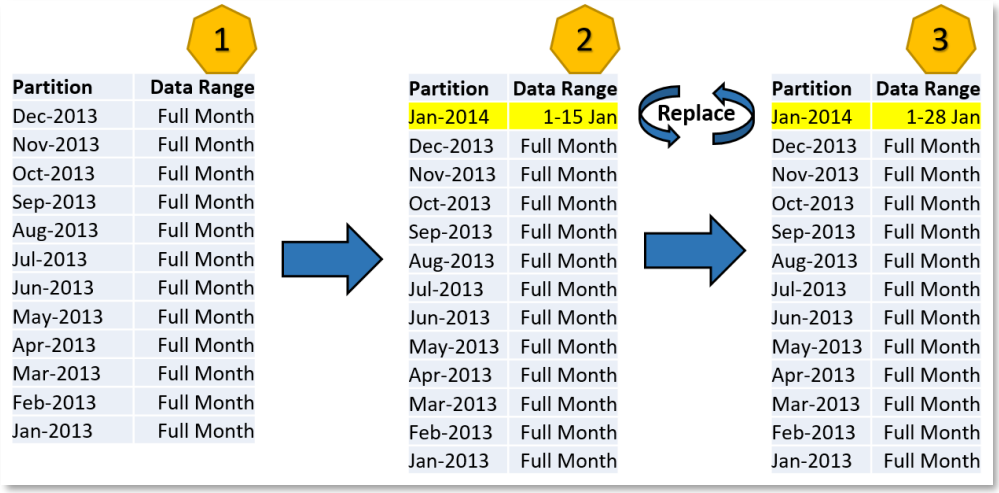
The execution plan is completely different than the first plan as parallelism is added at the partition level. Again, it reads data from the file (1 vertex), uses 1 vertex to work with each partition, and within each partition it loads data into 25 distributed buckets.


In the figure above, there are 13 partitions in the execution plan, and the 13th partition is the default or Catch-All partition, which does not get any data in this case (notice the rows count as 0) as all the data belongs to 2013 only.

Each INSERT statement generates a separate file in physical storage in Azure Data Lake Store, which can eventually lead to performance degradation. As a result, it is recommended to avoid small inserts and to rebuild a table after frequent insertions made with the ALTER TABLE <TableName> REBUILD command.

# Handling UPSERT

U-SQL tables do not support the UPDATE or DELETE statement, though similar functionality can be achieved by working with a partitioned table. Consider the partitioned table created earlier and into which data for year 2013 has been loaded (which should appear as does step 1 in the figure below).




Next, add a new partition for Jan-2014 and load partial data for just 15 days (01-Jan-2014 to 15-Jan-2014) with the below script (step 2 in the figure above):

```
@IncrementalDataINSERT =
    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_1Jan15Jan.csv"
    USING Extractors.Csv(skipFirstNRows : 1);

//Create a new empty partition if it doesn't exist
//For now we have hard-coded the partition assuming
//we have only one partition i.e. 201401
ALTER TABLE FactInternetSales_Partitioned ADD IF NOT EXISTS PARTITION ("201401");

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

Next, assume that you have another recent or updated file with Jan-2014 data starting from 10-Jan-2014 to 28-Jan-2014 (6 days' adjustments\modified data and new data starting from 16-Jan-2016). To load the recent data, drop the existing partition for Jan-2014 and add a new partition for Jan-2014, which will now contain old data from 01-Jan-2014 to 09-Jan-2014 and recent data from 10-Jan-2014 to 28-Jan-2014 (step 3 in the figure above).

The following script includes a step-by-step process for loading incremental data (both modified and new) into a partitioned U-SQL table.

```
//Create a rowset with all the recent data
//or incremental data
@IncrementalDataUPSERT =
    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_10Jan28Jan.csv"
    USING Extractors.Csv(skipFirstNRows : 1);
```

```
//Create a work table with data from all the partitions for which
//incremental data has been received
@WorkTableA =
SELECT DISTINCT CompleteData.*
    FROM FactInternetSales_Partitioned AS CompleteData
         INNER JOIN @IncrementalDataUPSERT
             AS IncrementalData
         ON CompleteData.CalendarYearMonth == IncrementalData.CalendarYearMonth;
//    SELECT CompleteData.*
//    FROM FactInternetSales_Partitioned AS CompleteData
//    WHERE CompleteData.CalendarYearMonth IN(SELECT CalendarYearMonth FROM
@IncrementalDataUPSERT);

//Remove rows for all the date for which we have recent data
//coming in the incremental data file from the above work table rowset
//This query assumes if there is any change in old data,
//incremental data contains all the data for that day including changes
//In case if you want go to the granular transaction level, you can add more
//conditions based on your need
@WorkTableMinusOldData = //WorkTableB
    SELECT WorkTableA.*
    FROM @WorkTableA AS WorkTableA
         LEFT OUTER JOIN
             @IncrementalDataUPSERT AS IncrementalData
         ON WorkTableA.OrderDateKey == IncrementalData.OrderDateKey
    WHERE IncrementalData.OrderDateKey == (int?) null;

//Union the unchanged data with the changed data
//coming as part of incremental data
@WorkTableWithIncrementalData = //WorkTableC
    SELECT *
    FROM @WorkTableMinusOldData  //WorkTableB
    UNION ALL
    SELECT *
    FROM @IncrementalDataUPSERT;

//Drop old partition and create a new empty partition
//For now we have hard-coded the partition assuming
//we have only one partition i.e. 201401
ALTER TABLE FactInternetSales_Partitioned DROP IF EXISTS PARTITION ("201401");
ALTER TABLE FactInternetSales_Partitioned ADD IF NOT EXISTS PARTITION ("201401");

//Load data for the impacted partitions
INSERT FactInternetSales_Partitioned ON INTEGRITY VIOLATION MOVE TO PARTITION
("209912")
SELECT * FROM @WorkTableWithIncrementalData; //WorkTableC
```

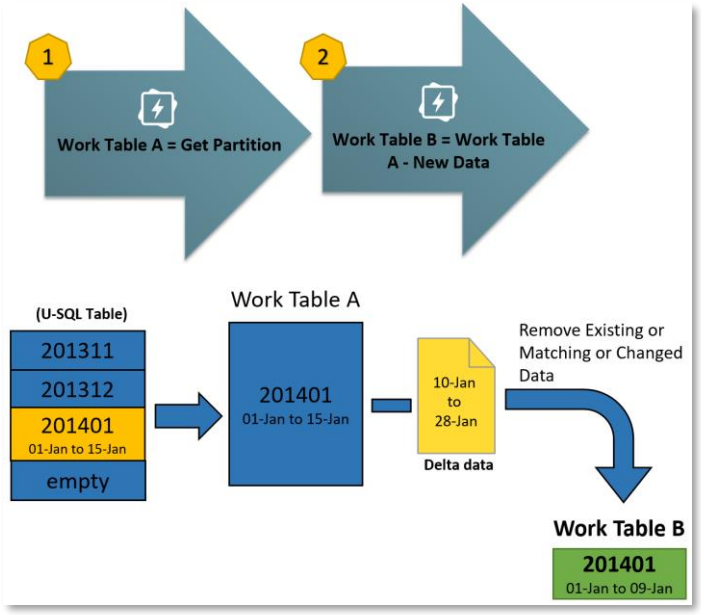
While the script above might look complex at first glance, a step-by-step walk through follows.

1. First, Work Table A is created. It contains data from the partitioned table for the months that you have received incremental data.

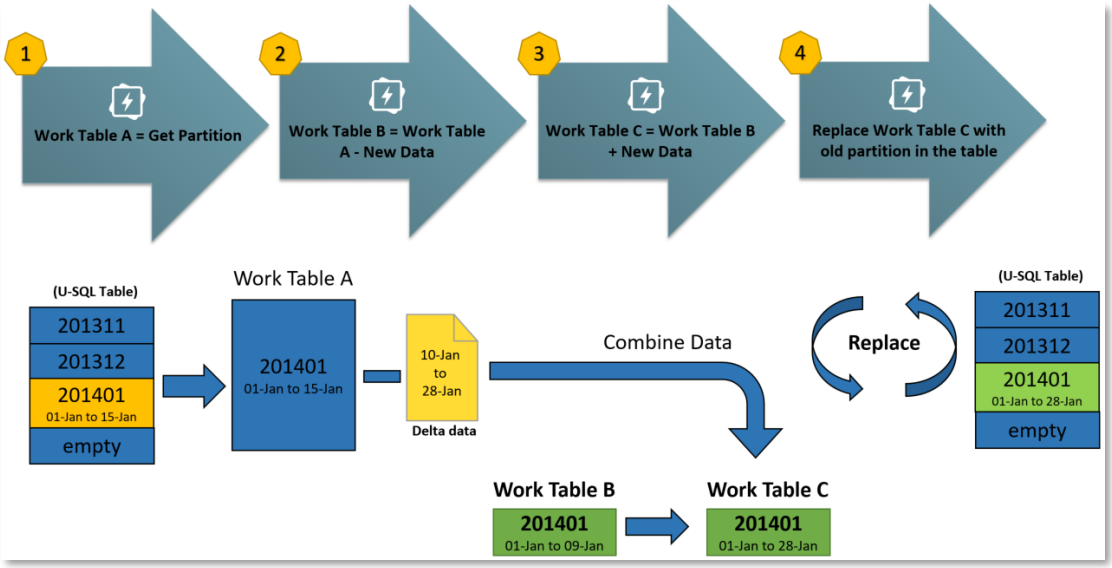
   *Please note, in the example there is a monthly partition, so the join is on month, though you can have daily or hour partitions as well based on your specific need.*

2. Next, Work Table B is created. It contains unchanged data from Work Table A. This is accomplished by joining incremental data with the data in Work Table A (for that specific partition) and filtering it based on matched data. Whatever is not matched is included in Work Table B.

   *The script above assumes that if there is any change in the data for a specific date, you receive all the data for that day (both changed and unchanged data). In some cases, you might receive only changed rows, in which case you can include a more granular joining condition, for example transaction number etc.*



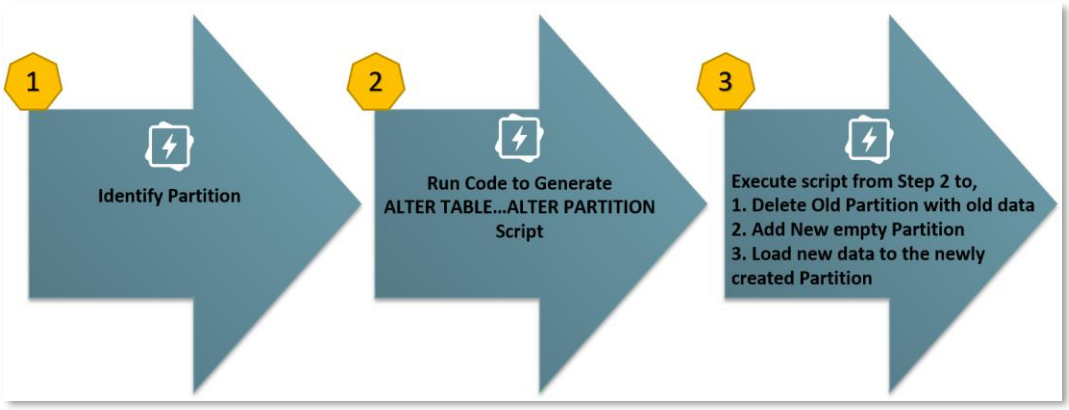

3. Next, Work Table C is created by combining Work Table B with incremental data by using the UNION statement.




Finally, the old partition is deleted from the table (which now includes changed data), a new empty partition is created, and data is loaded from Work Table C to the new partition.

# Dynamic partition management

In the previous script, the partition boundary for deleting and creating a new partition for the modified data was hard coded. This works well if you know the boundary value before your code execution, but that might not be the case in your production deployment. In this situation, you need a mechanism to identify the partitions (or partition boundaries) for which you have new set of modified data. To do this, you need to use the following steps:




## Step 1 – Identify partitions to work on

First, you need to identify the partition boundaries to work on from the incremental data. In the example, there is a monthly partition, so monthly boundaries are being identified from the incremental data with the below script (of course it will change based on your specific implementation of partition in your case, it could be yearly, monthly, daily, hourly etc.).

```
//Create a rowset with all the recent data
//or incremental data
@IncrementalDataUPSERT =
    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_10Jan28Jan.csv"
    USING Extractors.Csv(skipFirstNRows : 1);

//Select distinct Month(s) (partition boundary(ies))
@IdentifyPartitionsToWorkOn =
    SELECT DISTINCT CalendarYearMonth
    FROM @IncrementalDataUPSERT;

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

## Step 2 – Generate script based on template based on identified partitions – .NET code

After you identify partition periods to work on (and write it to a file), you can execute this C# procedure to generate ALTER TABLE statement for deleting impacted partitions and creating new empty partitions for those periods by reading the file created in step 1.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PartitionManagement
{
    static class PartitionManagement
    {
        public static void CreatePartitionScript(string InputCSVFileName, string
TableName, string OutScriptFileName)
        {
            StreamReader sr = new StreamReader(InputCSVFileName);
            StreamWriter sw = new StreamWriter(OutScriptFileName);
            string line;

            while ((line = sr.ReadLine()) != null)
            {
                var row = line.Split(',');

                sw.WriteLine("ALTER TABLE " + TableName + " DROP IF EXISTS PARTITION
(\"" + row[0] + "\");");
                sw.WriteLine("ALTER TABLE " + TableName + " ADD IF NOT EXISTS PARTITION
(\"" + row[0] + "\");");
            }
            sw.Close();
        }
    }
}
```

To call the C# procedure above, use the sample code below. You need to pass the location and name of the file that contains partition periods (generated in step 1), as well as the name of the destination U-SQL table. You also need to pass the file location and name of the file that will be produced using the procedure above with the ALTER TABLE statement for deleting impacted partitions and creating new empty partitions for those periods.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PartitionManagement
{
    class Program
    {
        static void Main(string[] args)
        {
            string csvfilepath =
@"D:\LearnUSQL\PartitionManagement\PartitionPeriods.csv";
            string tablename = @"FactInternetSales_Partitioned";
            string scriptfilepath =
@"D:\LearnUSQL\PartitionManagement\PartitionManageScript.txt";
            PartitionManagement.CreatePartitionScript(csvfilepath, tablename,
scriptfilepath);
        }
    }
}
```

## Step 3 – Delete and add new partitions with new datasets

Based on incremental data received, you have now identified impacted partitions. At this time, you can use a generated ALTER TABLE statement within your script as discussed in section 3.2.

# 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.