SQL Server to SQL Server PDW

Migration Guide (AU3)

Contents

4 Summary Statement

4 Introduction

- 4 SQL Server Family of Products
- 6 Differences between SMP and MPP
- 8 PDW Software Architecture
- 10 PDW Community

10 Migration Planning

- 11 Determine Candidates for Migration
- 11 Migration Considerations
- 12 Migration Approaches

13 SQL Server to PDW Migration

- 13 Main Migration Steps
- 13 PDW Migration Advisor
- 14 APS Migration Utility
- 14 Table Geometry
- 20 Optimization within PDW

29 Converting Database Schema Objects

29 Type Mapping

36 Syntax Differences

51 Conclusion

51 For More Information

51 Feedback

52 Appendix

© 2014 Microsoft Corporation. All rights reserved. This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes.

Summary Statement

In this migration guide you will learn the differences between the SQL Server and Parallel Data Warehouse database platforms, which runs on the Analytics Platform System appliance, and the steps necessary to convert a SQL Server database to Parallel Data Warehouse.

This guide has been updated to reflect the T-SQL improvements contained within Version 2, Appliance Update 3 (AU3)

Introduction

Migrating a data warehouse application from Microsoft® SQL Server® SMP to the Microsoft® SQL Server® Parallel Data Warehouse (PDW) region within the Microsoft Analytics Platform System (APS) appliance, provides many benefits, including improved query performance, scalability and the simplest integration to Hadoop.

This white paper explores the remediation activities and provides guidance to overcome technical differences between the two database platforms in order to simplify migration projects. It describes the implementation differences of database objects, SQL dialects, and procedural code between the two platforms.

SQL Server Family of Products

Microsoft offers a wide range of SQL Server products to accommodate the different business needs and technical requirements to provide a robust and scalable enterprise-wide solution.

Compact Edition

Microsoft SQL Server Compact 4.0 is a compact database ideal for embedding in desktop and web applications. SQL Server Compact 4.0 gives developers a common programming model with other SQL Server editions for developing both native and managed applications. SQL Server Compact provides relational database functionality in a small footprint.

Express Edition

SQL Server 2014 Express Edition is available for free from Microsoft and provides a powerful database engine ideal for embedded applications or for redistribution with other solutions. Independent software vendors use it to build desktop and data-driven applications. If you need more advanced database features and support for greater than 10 GB databases, SQL Server Express is fully compatible with other editions of SQL Server can be seamlessly upgraded to enterprise versions of SQL

Server without requiring remediation to overcome changes in functionality.

Standard Edition

SQL Server 2014 Standard edition is a robust data management and business intelligence database for departments and small workgroups to support a wide variety of applications. SQL Server Standard Edition also supports common development tools for on premise and cloud. Enabling effective database management with minimal IT resources. SQL Server Standard Edition is compatible with other editions.

Web Edition

SQL Server 2014 Web edition is a low total-cost-of-ownership option for Web hosters and Web VAPs to provide scalability, affordability, and manageability capabilities for small to large scale Web properties.

Business Intelligence Edition

SQL Server 2014 Business Intelligence edition delivers a comprehensive platform empowering organizations to build and deploy secure, scalable and manageable BI solutions. It offers exciting functionality such as browser based data exploration and visualization; powerful data mash-up capabilities, and enhanced integration management.

Enterprise Edition

SQL Server 2014 Enterprise edition delivers comprehensive high-end datacenter capabilities with blazing-fast performance, unlimited virtualization, and end-to-end business intelligence. Enabling high service levels for mission-critical workloads and end user access to data insights.

Parallel Data Warehouse

While the other versions of SQL Server mentioned here have a symmetric multi-processing (SMP) architecture, SQL Server Parallel Data Warehouse has a massively parallel processing (MPP) architecture, in the form of a data warehousing appliance, designed and built for to manage high volumes of relational data (with up to 100x performance gains) and providing the simplest integration to Hadoop. This addition of SQL Server only runs on the Microsoft Analytics Platform System appliance; available from HP, Dell and Quanta.

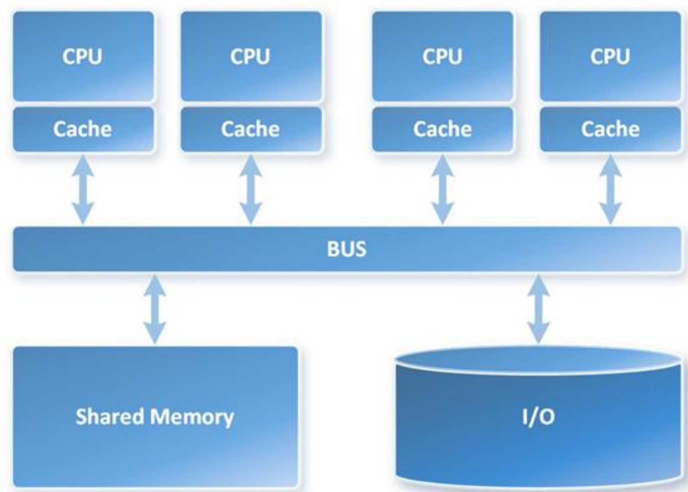
Differences between SMP and MPP

As data volumes grow and the number of users increase, we find the traditional architecture of running your data warehouse on a single SMP server insufficient to meet the business requirements. To overcome these limitations and scale and perform beyond your existing deployment, Parallel Data Warehouse brings Massively Parallel Processing (MPP) to the world of SQL Server. Essentially parallelizing and distributing the processing across multiple SMP compute nodes. SQL Server Parallel Data Warehouse is only available as part of Microsoft's Analytics Platform System (APS) appliance.

Before diving into the PDW architecture, let us first understand the differences between the architectures mentioned above.

Symmetric Multi-Processing (SMP)

Symmetric Multi-Processing (SMP) is the primary parallel architecture employed in servers. An SMP architecture is a tightly coupled multiprocessor system, where processors share a single copy of the operating system (OS) and resources that often include a common bus, memory and an I/O system.



Think of this as a typical single server, multi-core with locally attached storage, running the Microsoft Windows OS.

Massively Parallel Processing (MPP)

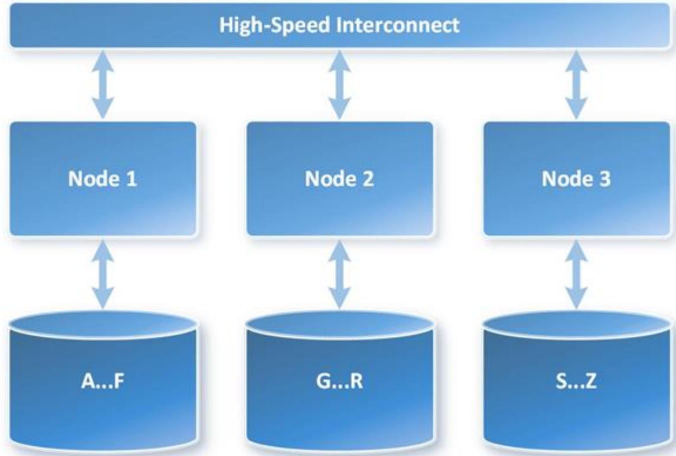
Massively Parallel Processing (MPP) is the coordinated processing of a single task by multiple processers, each working on a different part of the task. With each processor using its own operating system (OS) and memory. MPP processors communicate between each other using some form of messaging interface via an "interconnect". The setup for MPP is more complicated than SMP and one approach to simplify this while providing equal amounts of resources between processors is the "Shared-Nothing" architecture. This approach is the one Parallel Data Warehouse is based upon.

Shared Nothing Architecture

The term shared nothing architecture was coined by Michael Stonebraker (1986) to describe a multiprocessor database management system in which neither memory nor disk storage is shared among the processors.

For a database which follows the shared-nothing architecture, each processor has its own set of disks. Data is “horizontally partitioned” across nodes, such that each node has a subset of the rows from each table in the database. Each node is then responsible for processing only the rows on its own disks. Such architectures are especially well suited to data warehouse workloads, where large fact tables can be distributed across the nodes.

In addition, every node maintains its own lock table and buffer pool, eliminating the need for complicated locking and software or hardware consistency mechanisms. Because shared nothing does not typically have a severe bus or resource contention it can be made to scale to hundreds or even thousands of machines.



PDW Software Architecture

Admin Console

- The Admin Console is a web application that visualises the appliance state, health, and performance information.

MPP Engine

The MPP Engine is the brain of the SQL Server Parallel Data Warehouse (PDW) and delivers the Massively Parallel Processing (MPP) capabilities by doing the following:

- Generates the parallel query execution plan and coordinates the parallel query execution across the compute nodes.
- Stores and coordinates metadata and configuration data for all of the databases.
- Manages SQL Server PDW database authentication and authorization.
- Tracks hardware and software status.

Data Movement Service (DMS)

- Transfers data between the Computer nodes and the Control node.
- Processes query operations that require transferring data among the nodes.
- Improves query performance by optimizing data transfer speeds.
- Data is loaded in parallel directly from the loading server to the Compute nodes via the DMS.
- DMS transfers data from each Compute node directly to the backup server.
- Using PolyBase, DMS transfers data to and from an external Hadoop cluster, or the HDInsight Region on the appliance

SQL Server Databases

- Each Compute node runs an instance of SQL Server to process queries and manage user data.
- The Shell database manages the metadata for all distributed user databases.
- TempDB contains the metadata for all user temporary tables across the appliance.
- Master is the master table for SQL Server on the Control node

Configuration Tool

- The Configuration Tool (dwconfig.exe) is used by appliance administrators to configure the Analytics Platform System.

Domain Controller

- Performs authentication among the Analytics Platform System nodes, and manages the authentication of SQL Server PDW Windows Authentication logins
- Windows Domain Name Service (DNS) resolves domain names to IP addresses for the Analytics Platform System appliance.

Windows Deployment Service

- Windows Deployment Service (WDS) deploys the Windows Server operating system onto the appliance. It is deployed on every host and virtual machine across the appliance.
- The DHCP service creates IP addresses so that the hosts within the appliance domain can join the appliance network without having a pre-configured IP address.

Virtual Machine Manager

- Analytics Platform System uses virtualization to achieve high availability. The Virtual Machine Manager hosts System Center to deploy the operating system on the physical hosts.

- Windows Server Update Services (WSUS) provides the ability to apply or remove Windows Updates across all of the hosts and virtual machines.

PDW Community

Microsoft APS Documentation and Client Tools

The Microsoft Analytics Platform System product documentation and client tools can be obtained via the following locations:

- Microsoft Download Center

  http://www.microsoft.com/en-us/download/details.aspx?id=45294

Microsoft APS Instructor-Led Training

A number of different Microsoft Partners offer instructor-led training about Microsoft APS. Contact your Microsoft Account Manager to obtain a list of these partners within your region.

An online course is available on the Microsoft Virtual Academy at:

> http://www.microsoftvirtualacademy.com/training-courses/big-data-with-the-microsoft-analytics-platform-system

Microsoft Premier Support for APS

Microsoft Premier Support for APS is a service offered by Microsoft providing support through accredited appliance Microsoft Support Engineers, remote managed upgrades and update assistance.

Migration Planning

Before you begin the migration of an existing SQL Server data warehouse to SQL Server Parallel Data Warehouse (PDW) it is recommended that you first plan what it is you want to migrate and make a decision on the migration approach.

Determine Candidates for Migration

Migration of a data warehouse provides the perfect opportunity to remove data, objects and processes which are no longer utilized. Therefore the first step for migration planning is to identify and determine which objects and processes you wish to migrate.

- Identify candidate databases for migration
  - Primary Data Warehouses
  - Primary Data Marts
  - Staging Databases
  - Archive Databases
- Identify candidate objects for migration
  - Tables
  - Views
  - Stored Procedures
- Identify candidate data for migration
  - Archive non-required data

Migration Considerations

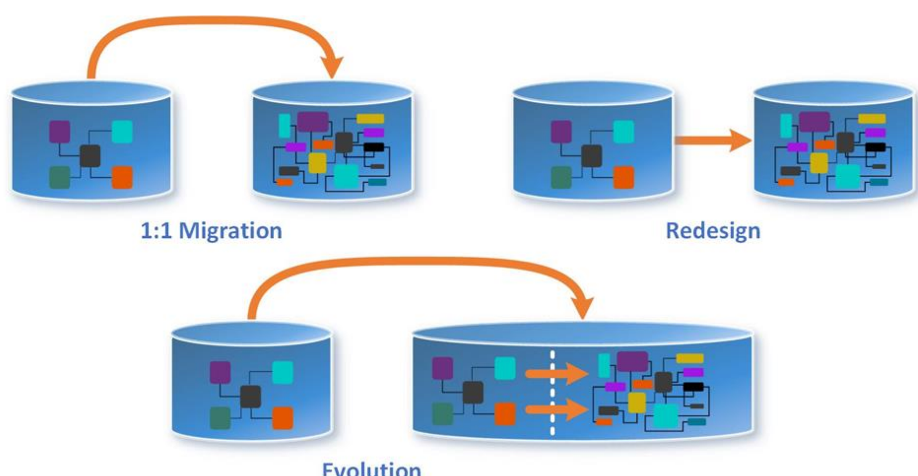
After determining candidates for migration, you should also consider the following areas and determine how you wish to manage them as part of the migration. For example, you may be planning to migrate multiple legacy SQL Server data marts onto a single SQL Server Parallel Data Warehouse, doing so may increase the importance of the APS appliance and therefore require an increase in availability and disaster recovery.

- Mission Critical
- High-Availability & Disaster Recover
- Security & Data Encryption
- Database Design
- Number of Users & Concurrency
- Design Complexities (ELT/ETL, UDT’s, UDF’s, Physical Storage)

- Data Loads (Volume, Frequency, Latency)
- Capacity Planning (current and future storage requirements)
- Backup & Restore

Migration Approaches

There are many different options to migrating data and applications from an existing SQL Server system to a SQL Server Parallel Data Warehouse, each of which provides different costs and benefits.



The three most widely adopted migration approaches are:

- 1:1 Migration – The existing data model and schema is moved directly to the SQL Server PDW with little or no change to the model. The benefits being the increasing performance, availability, scalability and the decreasing cost of ownership (TCO) of the platform. Sometimes referred to as Re-Hosting or the Forklift approach.
- Redesign – The data model is redesigned and the application is re-architected following the SQL Server PDW best practices. This can help reduce overall complexity and enable the ability for the data warehouse to be more flexible and answer any question at any time.
- Evolution – The evolutionary approach is a combination of the above two, allowing for the new target platform to be up and running quickly and over time deliver increasing benefits and capabilities by redesigning.

Determining the overall goal and reason for migration will provide the means to determine which of the 3 approaches is best suited.

Migration Drivers	Migration Approach
Performance	1:1 Migration or Redesign or Evolution
Availability	1:1 Migration or Evolution
Scalability	1:1 Migration or Evolution
Single View of the Business	Redesign or Evolution
Business Questions (Complexity)	Redesign or Evolution
Cost of Ownership (TCO)	1:1 Migration

SQL Server to PDW Migration

This section provides an overview of SQL Server to PDW migration. It discusses main migration steps, PDW Migration Advisor, table geometry, and optimization within PDW.

Main Migration Steps

This white paper provides you guidance in migrating the objects and highlights areas which are not supported within SQL Server PDW. To aid with identifying the migration areas to focus on, you can make use of the PDW Migration Advisor.

Once all objects have been migrated, the next step would be to migrate any integration or ETL jobs.

PDW Migration Advisor

The PDW Migration Advisor (PDWMA) is a tool that can be used to inspect your Microsoft SQL Server database schemas in order to display the changes that need to be made in order to migrate the database to Microsoft SQL Server PDW.

PDWMA is able to review the DDL and DML code structured and perform a series of validation checks to determine compatibility with PDW. This will significantly reduce the investigation efforts required in migration scoping projects. This whitepaper provides guidance in order to overcome technical issues that may be encountered in the database migration project.

A representative from the Microsoft technical pre-sales team will be happy to run the PDWMA for you – please connect with your Microsoft representative to run this tool/free service.

APS Migration Utility

The APS Migration Utility is a tool that can be used to automatically migrate the Microsoft SQL Server database schemas and data, whilst providing the ability to specifiy the table geometry during the migration process.

A representative from the Microsoft Consulting Services will be happy to run the APS Migration Utility for you – please connect with your Microsoft representative.

Table Geometry

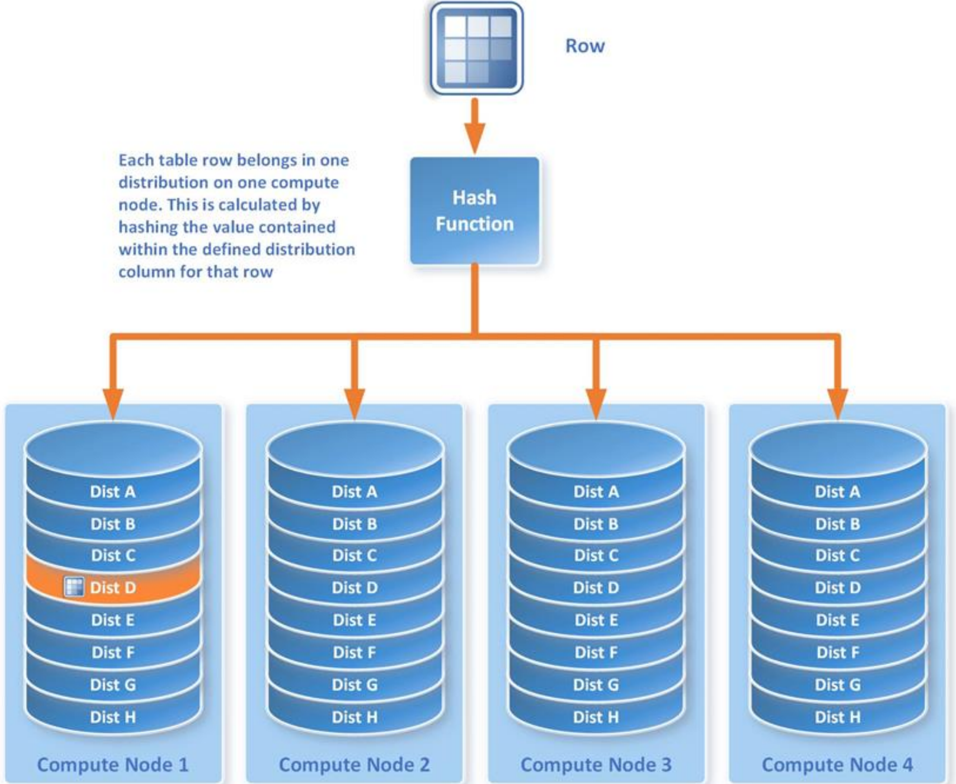
As we learnt within the previous sections, SQL Server Parallel Data Warehouse (PDW) is a Massively Parallel Processing (MPP) appliance which follows the Shared Nothing Architecture. This means that we need to distribute or spread the data across the compute nodes in order to benefit from the storage and query processing of the MPP architecture.

SQL Server PDW provides two options to define how the data can be partitioned, these are distributed and replicated tables.

Distributed Tables

A distributed table is a table in which all rows have been spread across the SQL Server PDW Appliance compute nodes based upon a row hash function. Each row of the table is placed on a single distribution, assigned by a deterministic hash algorithm taking as input the value contained within the defined distribution column.

The following diagram depicts how a row would typically be stored within a distributed table.



Each SQL Server PDW compute node has eight distributions and each distribution is stored on its own set of disks, therefore ring-fencing the I/O resources.

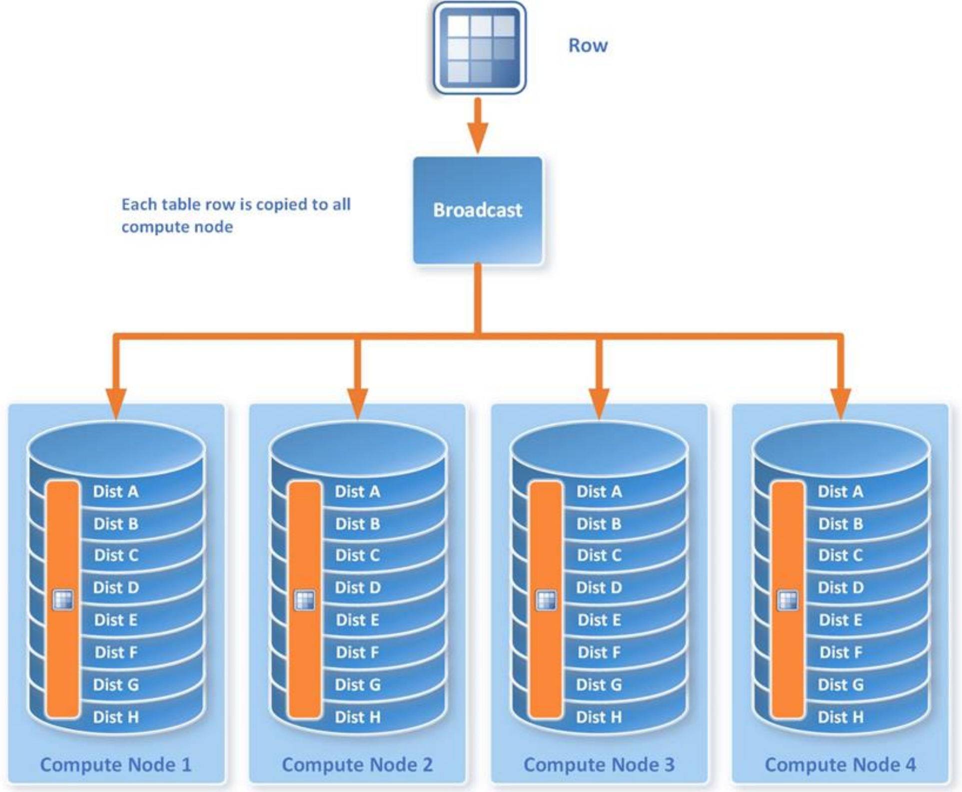
Distributed tables are what gives SQL Server PDW the ability to scale out the processing of a query across multiple compute nodes.

Each distributed table has one column which is designated as the distribution column. This is the column that SQL Server PDW uses to assign a distributed table row to a distribution.

When selecting a distribution column there are performance considerations, one of which is data skew. Skew occurs when the rows of a distributed table are not spread uniformly across each distribution. When a query relies on a distributed table which is skewed, even if a smaller distribution completes quickly, you will still need to wait for the queries to finish on the larger distributions. Therefore a parallel query performs as slow as the slowest distribution, and so it is essential to avoid data skew. Where data skew is unavoidable, PDW can endure 20 - 30% skew between the distributions with minimal impact to the queries.

Replicated Tables

A replicated table is a table in which a complete duplicate of all data is stored on each of the SQL Server PDW compute nodes. Replicating a table onto each compute node provides an additional optimization by removing the need to shuffle data between distributions before performing a join operation. This can be a huge performance boost when joining lots of smaller tables to a much larger tables, as would be the case within a type dimension data model, i.e. distribute the fact table on a high cardinality column, whilst replicating the dimensional tables.



The above diagram depicts how a row would be stored within a replicated table. A replicated table is striped across all of the disks assigned to each of the distributions within a compute node.

Because you are duplicating all data for each replicated table on each compute node, you will require extra storage, equivalent to the size of a single table multiplied by the number of compute nodes, therefore a table containing 100MB of data on a PDW appliance with 10 compute nodes will require 1GB of storage.

When loading new data, or updating existing data you will require far more resources to complete the task, than if it were to be executed against a distributed table. This is due to the fact that each operation will need to be executed on each compute node. Therefore it is essential that you take into account the extra overhead when performing ETL/ELT style operations against a replicated table.

The most frequently asked question about replicated tables would be “what is the maximum size we should consider for a replicated table?” We recommend that you keep the use of replicated tables to a small set of data, generally less than 5GB in size. However there are scenarios when you may want to consider much larger replicate tables. In which case we would recommend you follow one of these approaches to reduce the overall batch resource requirements.

- Maintain 2 versions of the same table, one distributed in which to perform all of the ETL/ELT operations against, and the second replicated for use by the BI presentation layer. Once all of the ETL/ELT operations have completed against the distributed version of the table, you would then execute a single CREATE TABLE AS SELECT (CTAS) statement in which to rebuild a new version of the final replicate table.

- Maintain 2 tables, both replicate tables. One “base” table persists all data, minus the current week’s set of changes. The second “delta” table persists the current week’s set of changes. A view is created over the “base” and the “delta” replicate tables which resolves the two sets of data into the final representation. A weekly schedule is required, executing a CTAS statement based upon the view to rebuild a new version of the “base” table, once complete the “delta” table can be truncated.

The Design Rules

A small number of database design choices can have a big impact on improving query performance. The following design rules will provide you with an outline to what information you should look for when making design decisions.

The key to achieving optimal performance using SQL Server Parallel Data Warehouse (PDW) is to make full use of the Massively Parallel Processing (MPP) architecture. You can achieve this by following two golden rules which are:

DISTRIBUTION FIRST

What we mean by “Distribution First” is that you should always begin with designing your data model to be distributed, as if you do not, you will not be able to realize the benefits of the Massive Parallel Processing (MPP) architecture. Therefore always begin with a distributed table.

Replicated tables should be viewed as an optimization technique, much in the same way you would view indexing or partitioning.

When designing your data model consider the following points:

- A single subject area within a data model should have a common distribution column which is also part of the join condition
- De-normalization with a Clustered ColumnStore Index as a way to overcome poor distribution column selection options.

MINIMIZE ALL MOVEMENT OF DATA

Data Movement comes in many forms such as:

- From physical disk to memory
- Redistribution of data between distributions as part of a SHUFFLE MOVE operation
- Duplication of data between compute nodes as part of a BROADCAST MOVE operation

We should attempt to minimize as much movement as possible to achieve the best possible performance. This can be achieved by better selection of distribution columns and optimization techniques.

Some data movement may be necessary to satisfy a query. It is worth mentioning that the PDW engine and Data Movement Service (DMS) does an excellent job of minimizing data movement operations using predicate pushdown, local/global aggregation and table statistics. So it is very important to keep PDW statistics up-to-date.

Distribution Column Selection Rules

Selecting a good distribution column is an important aspect to maximizing the benefits of the Massive Parallel Processing (MPP) architecture. This is possibly the most important design choice you will make in SQL Server Parallel Data Warehouse (PDW).

The principal criteria you must consider when selecting a distribution column for a table are the following:

- Access
- Distribution
- Volatility

The ideal distribution key is one that meets all three criteria. The reality is that you are often faced with trading one off from the other.

JOIN ACCESS - Select a distribution column that is also used within query join conditions

Consider a distribution column that is also one of the join conditions between two distributed tables within the queries being executed. This will improve query performance by removing the need to redistribute data via a SHUFFLE MOVE operation, making the query join execution distribution local.

AGGREGATE ACCESS – Select a distribution column that is also aggregation compatible

Consider a distribution column that is also commonly used within the GROUP BY clause within the queries being executed. This will improve query performance by removing the need to perform a two-step aggregation and redistribute data via a SHUFFLE MOVE operation, making the query GROUP BY operation execute distribution local.

DISTINCT ACCESS – Select a distribution column that is frequently used in COUNT DISTINCT's

Consider a distribution column that is commonly used within a COUNT DISTINCT function. This will improve query performance by removing the need to redistribute data via a SHUFFLE MOVE operation, making the query COUNT DISTINCT function execute distribution local.

DATA DISTRIBUTION – Select a distribution column that provides an even data distribution

Consider a distribution column that can provide an even number of rows per distribution, therefore balancing the resource requirements. To achieve this look for a distribution column that provides a large number of distinct values, i.e. at least 10 times the number of table distributions.

The data distribution of a table is said to be skewed if one or more distributions contain significantly more rows than others. Data skew is the enemy of all Massively Parallel Processing (MPP) systems as it puts the system into an unbalanced state which requires more overall resources within one or more distributions. When selecting a distribution column ensure that no single value represents a large percentage of rows (including NULL's). Unfortunately it is highly unlikely to be able to eliminate all data skew, but when selecting a distribution column, always remember, the greater the data skew, the greater the performance impact.

VOLATILITY – Select a distribution column that rarely, if ever changes value

PDW will not allow you to change the value of a distribution column for a given row by using an UPDATE statement. This is due to the fact that changing the value of a distribution column for a given row will almost certainly mean the row will be moved to a different distribution.

If you are required to change the value of a distribution column then you will need to first delete the existing row and then insert the new row. Alternatively you could use a CREATE TABLE AS SELECT (CTAS) statement to rebuild the table or partition.

It is therefore recommended that you select a distribution column which rarely requires the value to be modified.

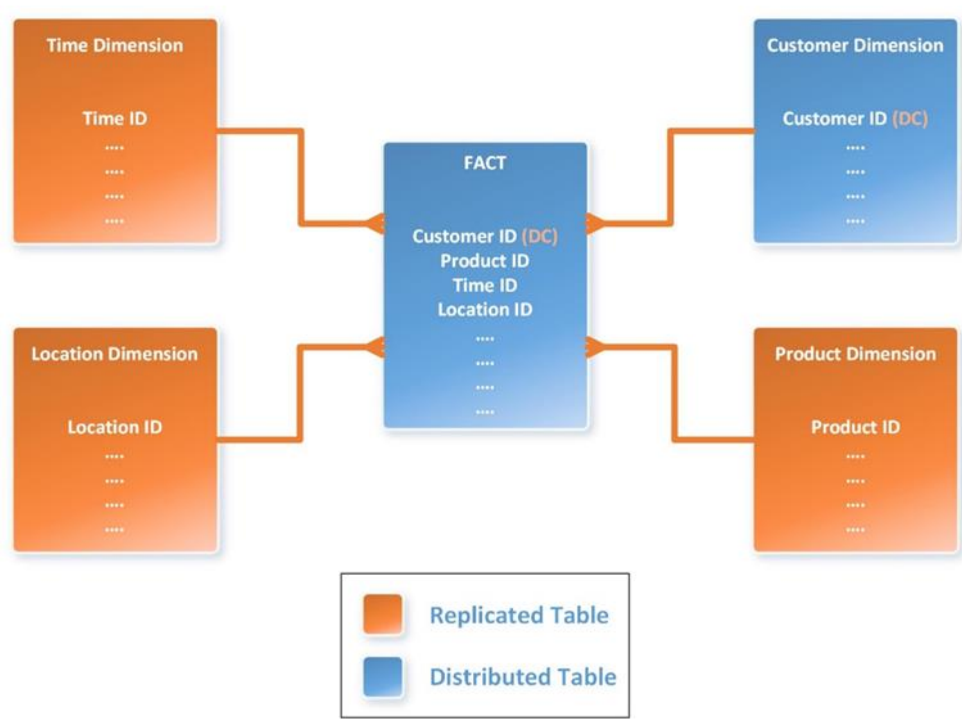
Optimization within PDW

SQL Server Parallel Data Warehouse (PDW) is a Massively Parallel Processing (MPP) appliance which provides you with large amounts of resources however these are still finite and overall query and concurrency performance will benefit from optimization of your data model using one or more of the following options.

Replicated Table

A replicated table is a table in which a complete duplicate of all data is stored on each of the SQL Server PDW compute nodes. Replicating a table can provide the ability for it to become distribution compatible when joined to another table. This removes the need to perform data movement via a SHUFFLE MOVE operation at the expense of data storage and load performance.

The ideal candidate for replicating a table is one that is small in size, i.e. less than 5GB, changes infrequently and has been proven to be distribution incompatible. Before selecting to replicate a table, ensure that you have exhausted all options for keeping the table distributed.



The dimensional model above depicts one fact table and four dimension tables. Within this example the customer dimension is significantly larger

in size than all other dimensions and based upon this would benefit remaining distributed. The “Customer ID” column provides good data distribution (minimal data skew), is part of the join condition between the fact table and the customer dimension creating a distribution compatible join, and is a regular candidate to be within the GROUP BY clause during aggregation. Therefore selecting “Customer ID” as the distribution column would be a good candidate. All other dimensions are small in size and to remove any need to redistribute the fact table when performing a join, we have made them replicated. Therefore eliminating all data movement when joining the fact table to any dimension table(s).

Clustered Indexes

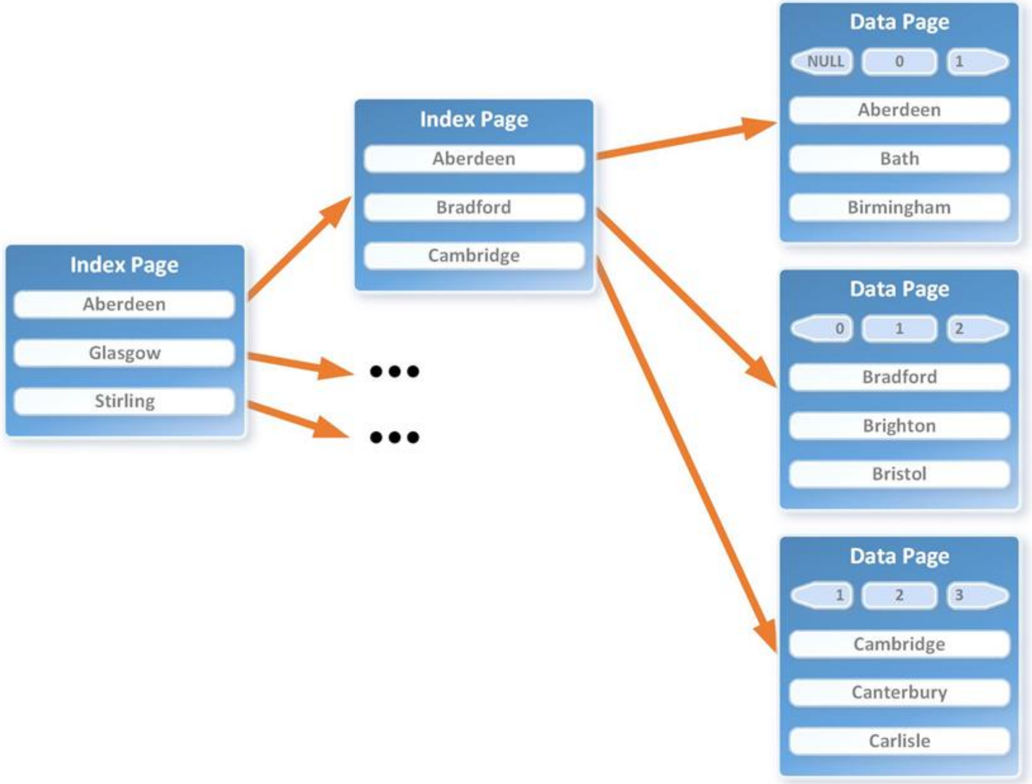
A Clustered Index physically orders the pages of the data in the table. If the table is distributed, then the physical ordering of the data pages is applied to each of the distributions individually. If the table is replicated, then the physical ordering of the pages is applied to each of replicate tables on each of the compute nodes.

All data pages within a clustered index table are linked to the next and previous data pages in a doubly linked list to provide ordered scanning. In other words, the records in the physical structure are sorted according to the fields that correspond to the columns used in the index.

PDW clustered index tables are always defined as non-unique, therefore each record will contain a 4-byte value (commonly known as an “uniquifier”) added to each value in the index where duplicate values exist.

You can only have one clustered index on a table, because the table cannot be ordered in more than one direction.

The following diagram shows what a Clustered Index table might look like for a table containing city names:



When defining a clustered index on a distributed table within PDW it is not necessary to include the distribution column within the cluster key. The distribution column can be defined to optimize the joins while the clustered index can be defined based upon the most frequently used columns as predicates or range operations.

Clustered indexes within PDW provides you with a way to optimize a number of standard OLAP type operations, including the following:

PREDICATE QUERIES

Data within the table is clustered on value and an index tree constructed for direct access to these data pages, therefore clustered indexes will provide the minimal amount of I/O required in which to satisfy the needs of a predicate query. Consider using a clustered index on column(s) commonly used as a valued predicate.

RANGE QUERIES

All data within the table is clustered and ordered on value, which provides an efficient method to retrieve data based upon a range query. Therefore consider using a clustered index on column(s) commonly used within a range predicate, for example where a given date is between two dates.

AGGREGATE QUERIES

All data within the table is clustered and ordered on value, which removes the need for a sort to be performed as part of an aggregation or a COUNT DISTINCT. Therefore consider using a clustered index on column(s) commonly contained within the GROUP BY clause or COUNT DISTINCT function.

Clustered Indexes should be selected over Non-Clustered Indexes when queries commonly return large result sets.

Non-Clustered Indexes

Non-Clustered Indexes are fully independent of the underlying table and up to 999 can be applied to both heap and clustered index tables. Unlike Clustered Indexes, a Non-Clustered Index is completely separate from the data, and on the index leaf page, there are pointers to the data pages. The pointer from the index to a data row is known as a row locator. The row locator is structured differently depending on whether the underlying table is a heap table or a clustered index table.

The row locator for a heap table will contain a pointer to the actual data page, plus the row number within the data page. If that row is updated and a data page split is required (if the updated row requires additional space), a forwarding pointer is left to point to the new data page and row. Over time and frequent updates can cause poor performance of the Non-Clustered Indexes, requiring maintenance via a rebuild of the underlying table and indexes.

The row locator for a clustered index table is different as it will contain the cluster key from the cluster index. To find a value, you first traverse the non-clustered Index, returning the cluster key, which you then use to traverse the clustered index to reach the actual data. The overhead to performing this is minimal as long as you keep the clustering key optimal. While scanning two indexes is more work than just having a pointer to the physical data page location, overall it can be considered better because minimal reorganization is required for any modification of the values in the table. This benefit is only true if the cluster key rarely, or never, changes.

The same reasons for selecting a clustered index are also true for a non-clustered index with the additional rules that they should only be used to optimize queries which return small result sets or when they could be utilized for covering indexes, reducing the base table IO requirements (However with the advent of Clustered ColumnStore Indexes this is less relevant).

NOTE: Non-Clustered Indexes can contribute to fragmentation and random IO, which goes against the sequential IO requirements of data warehousing. It is therefore recommended that you always begin with no Non-Clustered Indexes and add them only if it’s really necessary.

Clustered ColumnStore Indexes

A Clustered ColumnStore Index uses a technology called xVelocity for the storage, retrieval and management of data within a columnar data format, which is known as the columnstore. Data is compressed, stored, and managed as a collection of partial columns, which we call column segments.

Some of the clustered columnstore index data is stored temporarily within a rowstore table, known as a deltastore, until it is compressed and moved into the columnstore. The clustered columnstore index operates on both the columnstore and the deltastore, when returning the query results.

KEY CHARACTERISTICS

A clustered columnstore index in SQL Server Parallel Data Warehouse (PDW) has the following characteristics:

- Available in SQL Server PDW and above.
- Includes all columns in the table and is the method for storing the entire table.
- Fully updateable.
- Can be partitioned.
- It is the only index on the table, therefore cannot be combined with any other indexes.
- Uses columnstore compression which is not configurable.
- Does not physically store columns in a sorted order. Instead, it stores data to improve compression and performance. Pre-sorting of data can be achieved by creating the Clustered ColumnStore Index table from a Clustered Index table
- SQL Server PDW moves the data to the correct location (distribution, compute node) before adding it to the clustered columnstore index.
- For a distributed table, there is one clustered columnstore index for every partition of every distribution.
- For a replicated table, there is one clustered columnstore index for every partition of the replicated table on every compute node.

BENEFITS

SQL Server Parallel Data Warehouse (PDW) takes advantage of the column based data layout to significantly improve compression rates and query execution time.

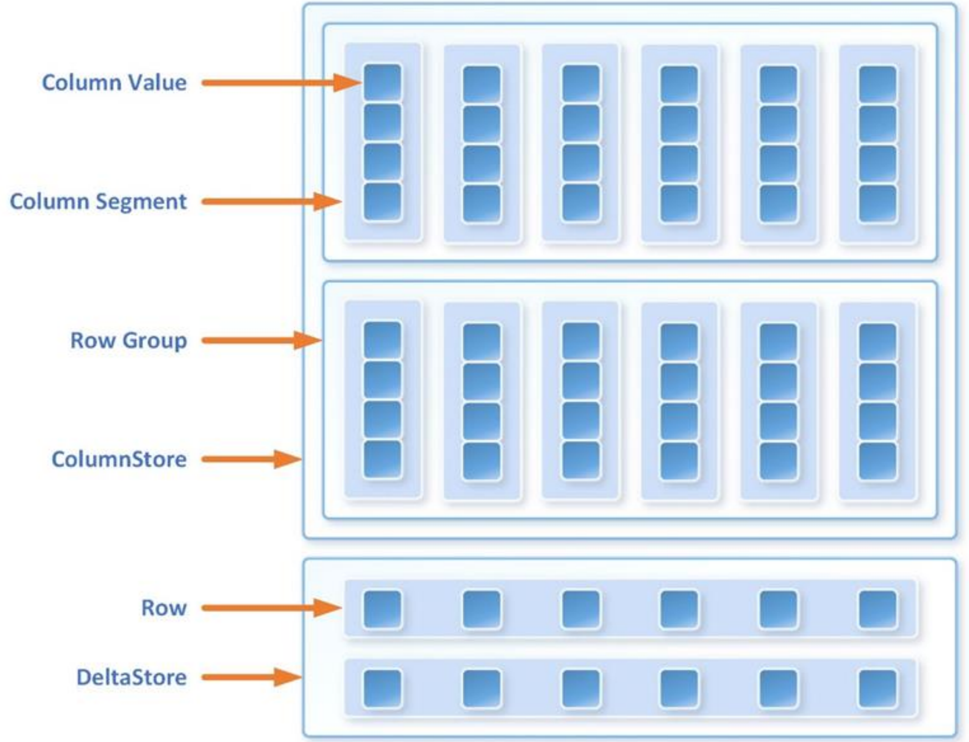
- Columns often have similar data, which results in high compression rates.
- Higher compression rates improve query performance by requiring less total I/O resources and using a smaller in-memory footprint.
- A smaller in-memory footprint allows for SQL Server PDW to perform more query and data operations in-memory.
- Queries often select only a few columns from a table, requiring less total I/O resources.
- Columnstore allows for more advanced query execution to be performed, by processing the columns within batches, which reduces CPU usage.

KEY TERMS

The following are key terms and concepts that you will need to know in order to better understand how to use clustered columnstore indexes.

- **Row Group** – is a group of rows that are compressed into columnstore format at the same time. Each column in the row group is compressed and stored separately on the physical media.
- **Column Segment** – a column segment is the basic storage unit for a columnstore index. It is a group of column values that are compressed and physically stored together on the physical media.
- **ColumnStore** – a columnstore is data that is logically organized as a table with rows and columns, physically stored in a columnar data format. The columns are divided into segments and stored as compressed column segments.
- **RowStore** – a rowstore is data that is organized as rows and columns, and then physically stored in a row based data format.
- **DeltaStore** – a deltastore is a rowstore table that holds rows until the quantity is large enough to be moved into the columnstore. When you perform a bulk load, most of the rows will go directly to the columnstore without passing through the deltastore. Some rows at the end of the bulk load might be too few in number to meet the minimum size of a rowgroup. When this happens, the final rows go to the deltastore instead of the columnstore.

- **Tuple Mover** – a background process which automatically compresses "CLOSED" Row Groups from the DeltaStore into Column Segments and storing them in the ColumnStore.



Partitioning

Partitioning allows you to physically divide up tables and indexes horizontally, so that a groups of data are mapped into individual partitions. Even though the table has been physically divided, the partitioned table is treated as a single logical entity when queries or updates are performed on the data.

BENEFITS

Partitioning large tables within SQL Server Parallel Data Warehouse (PDW) can have the following manageability and performance benefits.

- SQL Server PDW automatically manages the placement of data in the proper partitions.
- A partitioned table and its indexes appear as a normal database table with indexes, even though the table might have numerous partitions.
- Partitioned tables support easier and faster data loading, aging, and archiving. Using the sliding window approach and partition switching.
- Application queries that are properly filtered on the partition column can perform better by making use of partition elimination and parallelism.

- You can perform maintenance operation on partitions, efficiently targeting a subset of data to defragment a clustered index or rebuilding a clustered columnstore index.

SQL Server Parallel Data Warehouse (PDW) simplifies the creation of partitioned tables and indexes as you are no longer need to create a partition scheme and function. SQL Server PDW will automatically generate these and ensure that data is spread across the physical disks efficiently.

For distributed tables, table partitions determine how rows are grouped and physically stored within each distribution. This means that data is first moved to the correct distribution before determining which partition the row will be physically stored.

For clustered columnstore indexes, every partition will contain their own columnstore and deltastore. To achieve the best possible performance and maximise compression, it is recommended that you ensure each partition for each distribution is sized so that it contains more than 1 million rows.

Within SQL Server you are able to create a table with a constraint on the partitioning column and then switch that table into a partition. Since PDW does not support constraints, we do not support this type of partition switching method, but rather would require the source table to be partitioned with matching ranges.

Statistics Collection

SQL Server Parallel Data Warehouse (PDW) uses a cost based query optimizer and statistics to generate query execution plans to improve query performance. Up-to-date statistics ensures the most accurate estimates when calculating the cost of data movement and query operations.

SQL Server PDW stores two sets of statistics at different levels within the appliance. One set exists at the control node and the other set exists on each of the compute nodes.

COMPUTE NODE STATISTICS

SQL Server PDW stores statistics on each of the compute node, and uses them to improve query performance for the queries which execute on the compute nodes.

Statistics are objects that contain statistical information about the distribution of values in one or more columns of a table. The cost based query optimizer uses these statistics to estimate the cardinality, or number of rows, in the query result. These cardinality estimates enable the query optimizer to create a high-quality query execution plan.

The query optimizer could use cardinality estimates to decide on whether to select the index seek operator instead of the more resource-intensive index scan operator, improving query performance.

Each compute node has the AUTO_CREATE_STATISTICS set to ON, which causes the query optimizer to create statistics based upon a single column that is referenced within the WHERE or ON clause of a query. Multi-Column statistics are not automatically created.

CONTROL NODE STATISTICS

SQL Server PDW stores statistics on the control node, and uses them to minimize the data movement in the distributed query execution plan.

When you create statistics on a distributed table, SQL Server PDW creates a statistics object for each distribution table on each of the compute nodes, and one statistics object on the control node. The control node statistics object contains the result from aggregating the statistics object created for each of the distributions.

When you create statistics on a replicated table, SQL Server PDW creates a statistics object for each compute node, but since each compute node will contain the same statistics, the control node will only copy one statistics object from one compute node.

When you create statistics on an external table, SQL Server PDW will first import the required data into PDW so that it can then compute the statistics. The results are stored on the control node.

Statistics stored on the control node are not made available to client applications. The DMVs report only statistics on the compute nodes and so do not report statistics on the control node.

Collection of statistics can be performed based upon a default sample, custom sample percentage, filter selection or a full table scan. Increasing the amount of data sampled improves the accuracy of the cardinality estimates at the expense of the amount of time required in which to calculate the statistics.

Collection of Multi-Column statistics on all join columns within a single join condition and all aggregate columns within a GROUP BY clause will provide additional information for the optimizer to generate a better execution plan, such a density vectors.

Always ensure that statistics do not go stale, by updating the statistics automatically as part of the transformation schedule or a separate maintenance schedule, depending on the volatility of the column or columns.

As a minimum statistics should be created for each of the following:

- Distribution Column
- Partition Column
- Clustered Index Keys
- Non-Clustered Index Keys
- Join Columns
- Aggregate Columns
- Commonly used predicate columns

Converting Database Schema Objects

This section discusses migration issues that can be encountered when converting schema objects from SQL server to PDW and how they can be overcome.

Type Mapping

Migrating SQL Server Data Types is relatively easy as most data types are the same between SQL Server and PDW.

Equivalent Data Types

The following data types are the same between SQL Server and PDW.

```
binary, bit, char, date, datetime, datetime2,
datetimeoffset, decimal, float, int, bigint, money,
nchar, nvarchar, real, smalldatetime, smallint,
smallmoney, time, tinyint, varbinary, varchar
```

Data Types Not Supported

```
cursor, image, ntext, numeric, rowversion,
hierarchyid, table, text, timestamp, xml,
uniqueidentifier, Spatial Types, CLR data types
```

Data Type Migration Considerations and Mitigations

The following data types have differences or are not supported.

NUMERIC

The Numeric data type is the same as the Decimal data type in SQL server. In PDW the name "numeric" has been deprecated and PDW uses decimal instead. The way to convert from numeric to decimal is to change the definition name from numeric to decimal as the types are functionally equivalent.

```
CREATE TABLE dbo.Purchases
(
    PurchaseOrderID int NOT NULL,
    Value Numeric(9)
);
```

Translates to:

```
CREATE TABLE dbo.Purchases
(
    PurchaseOrderID int NOT NULL,
    Value Decimal(9)
);
```

DATE AND TIME FORMATS

The Date and Time data types are treated the same way in SQL Server and PDW. So data in PDW or SQL Server databases can be queried the same way and follows the same rules as far as both storage and syntax.

However it's possible that issues may arise when loading data. In many countries the date and time formats are different from the default of "ymd". In those cases the format needs to be handled in order to avoid loading errors via DWloader. DWloader is one of the ways to load data into PDW from text files. Text files however do not contain any type information and so it's possible that the format may be miss interpreted and cause errors forcing rows to be skipped (rejected) and logged during data loads. Dwloader.exe has numerous parameters in order to handle the type conversion into the appropriate format.

Example:

```
dwloader.exe –S MyPDW-SQLCTL01 -U mylogin -P 123jkl –
i file.txt -t "|" -r \r\n –D "dmy"
```

The date format in this case assumes that all the columns have the date values ordered as day (d), month (m), year (y).   It's also possible that formats differ between columns. In that case a format file will have to be used that specifies the input format on a per column basis. The format file can be specified with the –dt parameter.

```
dwloader.exe –S MyPDW-SQLCTL01 -U mylogin -P 123jkl –
i file.txt -t "|" -r \r\n –dt inputformatfile.txt
```

The contents of the inputformatfile.txt would look like this. Each line has the name of the column and the format of the destination datetime data type.

```
LastReceiptDate=ymd
ModifiedDate=mdy
```

When importing text data it's also possible that due to errors in the source data the date format for a specific column varies from row to row and this can either lead to errors or loading incorrect data. There may also be corruption errors in the input data that need to be detected. In such cases dwloader.exe offers some options to detect and log the rows that caused a load error. This would then allow bayou to remediate the data and reload.

In these cases the following dwloader parameters are useful. The complete list of dwloader parameters is available in the product books online.

-rt { value | percentage }
Specifies whether the -reject_value in the -rv reject_value option is a literal number of rows (value) or a rate of failure (percentage). The default is value. The percentage option is a real-time calculation that occurs at intervals according to the -rs option. For example, if the Loader attempts to load 100 rows and 25 fail and 75 succeed, then the failure rate is 25%.

-rv reject_value
Specifies the number or percentage of row rejections to allow before halting the load. The -rt option determines if reject_value refers to the number of rows or the percentage of rows. The default reject_value is 0. When used with -rt value, the loader stops the load when the rejected row count exceeds reject_value. When use with -rt percentage, the loader computes the percentage at intervals (-rs option). Therefore, the percentage of failed rows can exceed reject_value.

-rs reject_sample_size
Used with the -rt percentage option to specify the incremental percentage checks. For example, if reject_sample_size is 1000, the Loader will calculate the percentage of failed rows after it has attempted to load 1000 rows. It recalculates the percentage of failed rows after it attempts to load each additional 1000 rows.

-R load_failure_file_name
If there are load failures, dwloader stores the row that failed to load and the failure description the failure information in a file named load_failure_file_name. If this file already exists, dwloader

will overwrite the existing file. load_failure_file_name is created when the first failure occurs. If all rows load successfully, load_failure_file_name is not created.

-e character_encoding
Specifies a character-encoding type for the data to be loaded from the data file. Options are ASCII (default), UTF8, UTF16, or UTF16BE, where UTF16 is little endian and UTF16BE is big endian. These options are case insensitive.

-m
Use multi-transaction mode for the second phase of loading; when loading data from the staging table into a distributed table.

With –m, SQL Server PDW performs and commits loads in parallel. This performs much faster than the default loading mode, but is not transaction-safe.

Without –m, SQL Server PDW performs and commits loads serially across the distributions within each Compute node, and concurrently across the Compute nodes. This method is slower than multi-transaction mode, but is transaction-safe.

-m is optional for append, reload, and upsert.

-m is required for fastappend.

-m cannot be used with replicated tables.

-m applies only to the second loading phase. It does not apply to the first loading phase; loading data into the staging table.

There is no rollback with multi-transaction mode, which means that recovery from a failed or aborted load must be handled by your own load process.

We recommend using –m only when loading into an empty table, so that you can recover without data loss. To recover from a load failure: drop the destination table, resolve the load issue, re-create the destination table, and run the load again.

TIMESTAMP
Timestamp in PDW is treated differently to SQL Server. In SQL server the current practice is to use the **rowversion** keyword instead of the **timestamp** keyword however the two commands are functionally equivalent in SQL Server. In PDW there is no direct functional equivalent to the **rowversion** that auto updates if the row is changed. In PDW partially similar functionality can be achieved by using the datetime2

data type and **CURRENT_TIMESTAMP**. Note that in PDW the column does not auto update if the row is changed like it does in SQL Server.

In SQL Server **rowversion** and **timestamp** columns are used to provide application controlled row change detection and optimistic locking for mainly OLTP databases. In PDW the same is not true since the platform is designed to support data warehouse requirements rather than thousands of singleton insert and updates.

The issue can sometimes arise when data needs to be migrated to the data warehouse and a column is needed to contain the source **rowversion** data. The binary data type can provide a solution here.

- Non nullable **rowversion / timestamp** data type is equivalent to BINARY(8) data type.

- Nullable **rowversion / timestamp** data type is equivalent to VARBINARY(8) data type.

If this functionality is required then either the application will need to track and make the relevant updates in application logic or this functionality could be undertaken in SQL Server and the resulting data loaded into PDW.

For example:

```
CREATE TABLE Table1 (Key1 INT, VersionCol
ROWVERSION);
```

Can be converted to the following in PDW:

```
CREATE TABLE Table1 (Key1 INT, VersionCol BINARY(8));
```

There are also cases where a timestamp is needed in order to track a certain action or event or even provide for some rudimentary application based locking behavior. Although it is not a best practice to carry out OLTP style workloads within a data warehouse there may be no other easy option to overcome a specific issue. A common use of this is to be able to easily record the batches of data loaded into the data warehouse or a specific multistep ELT process.

For example:

```
CREATE TABLE ProcessRunning
(
   BatchID INTEGER,
   DescriptionVal VARCHAR(30),
   col2 DATETIME2
);

DECLARE @currTS DATETIME2
```

```
SET @currTS = CURRENT_TIMESTAMP
INSERT INTO ProcessRunning VALUES (1, 'BIG ELT Xform
process', @currTS);
-- The BatchID could then be used to identify added
rows in a given batch.
```

MIGRATING TEXT, BLOBS AND VARCHAR(MAX), NVARCHAR(MAX), VARBINARY(MAX)

The following data types are not supported in PDW:

- TEXT
- VARCHAR(MAX)
- NVARCHAR(MAX)
- VARBINARY(MAX)
- XML

When dealing with these data types there are several options that need to be evaluated.

1. Leave the large BLOB data in the source operational systems.
2. Break up BLOB data into logical fields and then migrate. In many cases VARCHAR(MAX) columns are being used to store data that could be broken down further into different columns and different data types. While this is not always possible a simple investigation of the actual data in the VARCHAR(MAX) fields can yield a lot of information and provide more choices. When following this approach you also need to consider the row size limitations.
3. Determine the actual size of the largest amount data in the VARCHAR(MAX) field. In most cases this is a small enough value that the data could be translated to a smaller data type.

For example:

```
-- Displays the max length of the largest string.
SELECT MAX(DATALENGTH(DescriptionVal)) FROM tbl1
```

If the size of the data is still too big to fit in the row and including it would exceed the 8KB per page limit then another potential option is to have the large text column in a separate table and related to the main row by an ID.

Migrating or Working with Spatial Data Types

In the current release PDW does not support directly performing calculations with spatial data types in the same way as is possible with SQL Server.   Having said this it is possible to store spatial data in PDW so that it can be retrieved and processed in a seamless fashion in order perform further analysis by utilizing a distributed database architecture. In this case PDW would be the master system hosting the source raw data. SQL server hosts could then extract and post process this data to perform other calculations. Fast Infiniband interconnects provide sufficient throughput to transfer large volumes of data at a high speed in order to both reduce the burden on the central enterprise data warehouse as well as harness native functionality that is available in SQL server today.

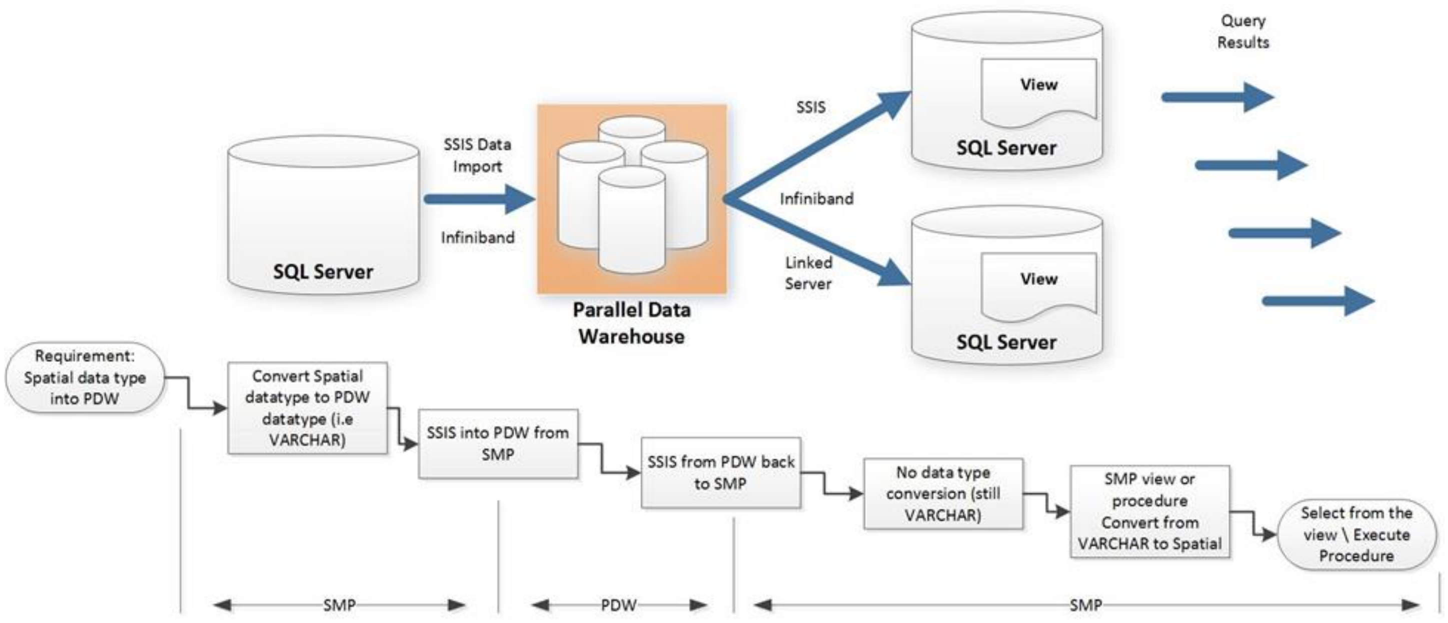
Base raw data can be migrated from SQL Server and stored in existing PDW data types with no loss. The data can be retrieved via normal T-SQL queries and the spatial processing can be done via SQL server.

Importing Data - Spatial Data can be converted into data which can then be stored within a PDW compatible data type such as an NVARCHAR() or VARCHAR(). Importing Spatial Types and converting them to VARCHAR can be accomplished via SSIS, CLR or custom data conversion application/script.

Retrieving Data - SQL Server can connect to PDW via a linked server connection, querying data directly or retrieving data via SSIS. Both connection types operate over the fast Infiniband network to support fast data transfers.

Linked servers can be queried within SQL server in much the same fashion as tables. A view would then be created to convert back to a spatial type to support geometric calculations within SQL Server. Overall this solution would be cost effective to implement since the heavy duty data retrieval and aggregations work is done by PDW in a scalable fashion. Geospatial calculations and any post processing can be performed on less expensive SMP servers running SQL Server Standard Edition.

This is an example of working with spatial data.



Syntax Differences

There are syntax differences to be aware of between SQL Server and PDW. In most cases the extra complexity needed in the SQL Server syntax has been removed as it's not applicable for PDW. This reduced syntax leveraging the appliance model is both easier to learn as well as less complex to support in the long term.

Create Database Statement

The create database statement syntax is significantly simpler than the syntax needed for SQL Server.

```
CREATE DATABASE database_name
WITH (
    [ AUTOGROW = ON | OFF , ]
    REPLICATED_SIZE = replicated_size [ GB ] ,
    DISTRIBUTED_SIZE = distributed_size [ GB ] ,
    LOG_SIZE = log_size [ GB ] )
[;]
```

The appliance automatically places the database files on the appropriately for best performance and therefore the file specification and other file related parameters are not needed. This paper has already covered concepts relating to distributed tables vs replicated tables in previous sections.

REPLICATED_SIZE
The amount of data in GB allocated for replicated tables. This is the amount of data allocated on a per compute node basis.

DISTRIBUTED_SIZE
The amount of data in GB allocated for distributed tables. This is the amount of data allocated over the entire appliance. Meaning that in a on a per compute node basis the disk space allocated equals the DISTRIBUTED_SIZE divided by the number of compute nodes in the appliance.

LOG_SIZE
The amount of data in GB allocated for the transaction log. This is the amount of data allocated over the entire appliance. Meaning that in a on a per compute node basis the disk space allocated equals the LOG_SIZE divided by the number of compute nodes in the appliance.

AUTOGROW
Governs if the database size is fixed or if the appliance will automatically increase the disk allocation for the database as needed until the amount of physical disk space in the appliance is consumed. It is recommended that the database be created big enough before loading data in order to minimize auto grow events that can impact data loading performance.

MIGRATING A SQL SERVER DATABASE AND SIZING FOR PDW

When migrating databases one must calculate the amount of disk space allocated for replicated vs distributed tables. These general rules provide guidance for when to make a table replicated or distributed.

- When to set tables as replicated.
  - Dimension and reference tables are typically replicated.
  - Tables with less than 5GB of heap data are typically replicated.
  - Tables that have no identifiable distribution key that also match the other criteria.
- When to set tables as distributed.
  - Fact tables should be distributed.
  - Tables with greater than 5 GB of heap data (excluding indexes) is considered large enough to typically be set as distributed.

– Tables that have columns with reasonably high cardinality and where these columns are ideally also present in aggregation clauses in queries submitted.

– Understanding the business and data will also highlight distributed tables.

Once the table sizes are known and categorized then it's a simple matter of adding a percentage for growth on a year to year basis. It is also important to allow sufficient space to be able to make a copy of the largest distributed table should the need arise. If partitioning is used on the fact table(s) since its so large then sufficient space for at least a few extra partitions should be allocated. These simple guidelines together with the categorization of tables between distributed vs. replicated will provide sufficient insight into sizing the database.

Create Table Statement

The main differences in syntax relating to the create table statement is the extra syntax related to nominating a table as either replicated or distributed or if the table is stored as a clustered column store.

For example:

```
CREATE TABLE myTable
(
    id INT NOT NULL
  , lastName VARCHAR(20)
  , zipCode VARCHAR(6)
)
WITH
(
    DISTRIBUTION = HASH (id)       -- <- Nominates the
distribution column
  , CLUSTERED COLUMNSTORE INDEX    -- <- Specifies
column store.
);
```

Replicated tables are defined with this syntax:

```
WITH
(
    DISTRIBUTION = REPLICATE       -- <- Nominates the
TABLE as replicated
);
```

Create Table Differences

- PRIMARY KEY and UNIQUE syntax is not supported.
- Data compression is always page compression unless the table is created with a column store structure.
- FILESTREAM storage is not supported.
- Cannot specify that a table is to be created in a specific filegroup by naming the filegroup. Partitions can be used to spread and segment data. This reduces the effort required by the DBA.
- Referential FOREIGN KEY constraints cannot be defined. However default constraints are supported. In data warehouse designs referential integrity constraints impose a very significant performance penalty when loading data and updating data as the constraints have to be validated. Current practice with enterprise data warehouse systems is to avoid the use of referential constraints in favor of higher performance.
- Cascade update and delete operations are not supported.

Specifying Partitions

Partitions must be specified as part of the create table statement. The partition scheme and partition function syntax that is required for SQL Server is not required for PDW. The partition functions and schemes are setup automatically by PDW.

For example:

The following is SQL Server Syntax:

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000)
;
GO

CREATE PARTITION SCHEME myRangePS1
AS PARTITION myRangePF1
TO (test1fg, test2fg, test3fg, test4fg)
;
GO

CREATE TABLE myTable
(
    ID INTEGER,
    Col1 VARCHAR(10),
    Col2 VARCHAR(10)
)
ON myRangePS1 (ID)
;
```

Translates to the following PDW Syntax (assuming a distributed table):

```
CREATE TABLE myTable
(
    ID INTEGER,
    Col1 VARCHAR(10),
    Col2 VARCHAR(10)
)
WITH
(
    DISTRIBUTION = HASH(ID)
  , PARTITION (ID RANGE LEFT FOR VALUES (1, 100,
1000))
);
```

The $PARTITION function is not supported in queries however partition elimination is still supported and performed via the use of WHERE clause predicates.

For example:

The following is SQL Server Syntax:

```
SELECT COUNT(*) FROM myTable
WHERE col1 = "xx"
  AND $PARTITION.myRangePF1(ID) = 2
;
```

Translates to the following PDW Syntax:

```
SELECT COUNT(*) FROM myTable
WHERE col1 = "xx"
  AND ID > 100 AND ID <= 1000   -- Specifies
partition 2
;
```

Migrating Triggers

PDW does not support Triggers functionality in the current release. Triggers are mainly used in OLTP solutions and therefore have very limited application in data warehouse solutions. Triggers in OLTP solutions are used for implementing cascading update functionality or other logic based on changed rows. In data warehouse solutions data changes can be implemented via operational data stores (ODS) or in the ETL/ELT processes that cleanse and layout data for maximum retrieval query and processing performance.

It is recommended that functionality requiring triggers is implemented in ODS systems and not in the enterprise data warehouse. A hub and spoke architecture using both PDW and SQL Server would be an ideal solution to such issues. Intermediate work can be performed on SMP SQL servers and the heavy duty queries and aggregations can be executed against PDW.

Schema Ownership

Schemas operate in a similar fashion in PDW as they do in SQL Server. In SQL Server when creating a schema within the same transaction you can also create tables, views, and set GRANT, DENY or REVOKE permissions on those objects. This is not supported in PDW.

SQL Server CREATE SCHEMA:

```
USE AdventureWorks2012;
GO
CREATE SCHEMA Sprockets AUTHORIZATION mwinter
    CREATE TABLE NineProngs
    (
        source int
      , cost int
      , partnumber int
    )
    GRANT SELECT ON SCHEMA::Sprockets TO julius
    DENY SELECT ON SCHEMA::Sprockets TO danny;
GO
```

Equivalent PDW Syntax:

```
USE AdventureWorks2012;
CREATE SCHEMA Sprockets AUTHORIZATION mwinter
GO
CREATE TABLE Sprockets.NineProngs
(
    source int
  , cost int
  , partnumber int
)
WITH (DISTRIBUTION = HASH(source))
GRANT SELECT ON SCHEMA::Sprockets TO julius
DENY SELECT ON SCHEMA::Sprockets TO danny
GO
```

Views

Views operate in a similar fashion in PDW as they do in SQL Server. In PDW the view definitions are stored in the control node and expanded during query compilation. As such schema binding and materialized views are not supported.

Unsupported View Creation Options:

```
WITH CHECK OPTION, ENCRYPTION, SCHEMABINDING,
VIEW_METADATA
```

Migrating SELECT Statements

The select statement between SQL Server and PDW is very similar. Minor additions exist in order to harness the added performance that PDW can provide when selecting and transforming large amounts of data. PDW introduces a new statement called Create Table as Select (CTAS). This

replaces the SQL Server SELECT INTO statement and provides additional functionality, such as the ability to specify the table geometry (Distributed, Replicate), indexing (Heap, Clustered Index, Clustered ColumnStore Index) and partitioning options. This operation is also minimal logged by default.

Changing SELECT INTO syntax to CTAS Statements will leverage the added performance that PDW provides. The CTAS statement is covered in other sections of this document so this example focuses on the syntax differences.

For example, the SQL Server SELECT INTO Statement:

```
SELECT
    c.ID
  , c.FirstName
  , c.LastName
  , e.JobTitle
  , a.AddressLine1
  , a.City
  , a.PostalCode
INTO dbo.AddressesList
FROM Person.Person AS c
  INNER JOIN HumanResources.Employee AS e
    ON  e.BusinessEntityID = c.BusinessEntityID
  INNER JOIN Person.BusinessEntityAddress AS bea
    ON  e.BusinessEntityID = bea.BusinessEntityID
  INNER JOIN Person.Address AS a
    ON  bea.AddressID = a.AddressID
  INNER JOIN Person.StateProvince as sp
    ON  sp.StateProvinceID = a.StateProvinceID
```

Translates to the following PDW statement (assuming the table is distributed on the ID column):

```
CREATE TABLE dbo.AddressesList
WITH (DISTRIBUTION = HASH (ID))
AS
SELECT
    c.ID
  , c.FirstName
  , c.LastName
  , e.JobTitle
  , a.AddressLine1
  , a.City
  , a.PostalCode
FROM Person.Person AS c
  INNER JOIN HumanResources.Employee AS e
    ON  e.BusinessEntityID = c.BusinessEntityID
  INNER JOIN Person.BusinessEntityAddress AS bea
    ON  e.BusinessEntityID = bea.BusinessEntityID
  INNER JOIN Person.Address AS a
    ON  bea.AddressID = a.AddressID
  INNER JOIN Person.StateProvince as sp
    ON  sp.StateProvinceID = a.StateProvinceID
```

Assuming a replicated table, the translation would be:

```
CREATE TABLE dbo.AddressesList
WITH (DISTRIBUTION = REPLICATE)
AS
SELECT
    c.ID
  , c.FirstName
  , c.LastName
  , e.JobTitle
  , a.AddressLine1
  , a.City
  , a.PostalCode
FROM Person.Person AS c
  INNER JOIN HumanResources.Employee AS e
    ON  e.BusinessEntityID = c.BusinessEntityID
  INNER JOIN Person.BusinessEntityAddress AS bea
    ON  e.BusinessEntityID = bea.BusinessEntityID
  INNER JOIN Person.Address AS a
    ON  bea.AddressID = a.AddressID
  INNER JOIN Person.StateProvince as sp
    ON  sp.StateProvinceID = a.StateProvinceID
```

The body of the select statement remains the same within the PDW syntax translation. All that has changed is the addition of the table creation portion of the syntax. These relatively minor differences in syntax can provide orders of magnitude better performance between SMP and MPP platforms.

Migrating UPDATE Statements

The update statement in PDW is similar to the update statement functionality in SQL Server in most ways. However there are some differences that will require modification of update statement syntax.

- Update statements cannot contain the TOP clause.
- Update statements cannot contain a FROM clause and a sub-query. For example, the following statement is not allowed:

```
UPDATE r
SET update_ts = CURRENT_TIMESTAMP
FROM region AS r
WHERE EXISTS (SELECT 1
              FROM nation AS n
              WHERE n.region_key = r.region_key
                AND n.nation_key = 0)
```

  The above query can be rewritten as an implicit join:

```
UPDATE region
SET update_ts = CURRENT_TIMESTAMP
FROM nation AS n
WHERE n.region_key = region.region_key
  AND n.nation_key = 0
```

- Update statements cannot contain explicit joins in the FROM clause. For example, the following statement is not allowed:

```
UPDATE c
SET population_ct = population_ct + 1
FROM census AS c
  INNER JOIN location AS l
    ON c.location_key = l.location_key
WHERE l.country_nm = 'Australia'
  AND l.state_nm = 'Victoria'
```

  The above query can be rewritten as an implicit join in the FROM clause. The following statement is allowed:

```
UPDATE census
SET population_ct = population_ct + 1
FROM location AS l
WHERE census.location_key = l.location_key
WHERE l.country_nm = 'Australia'
  AND l.state_nm = 'Victoria'
```

- Update statements cannot contain more than one join in the FROM clause. For example, the following statement is not allowed:

```
UPDATE order_line
SET dispatch_ts = CURRENT_TIMESTAMP
FROM customer AS c
   , order AS o
WHERE order_line.order_key = o.order_key
  AND order_line.customer_key = c.customer_key
  AND o.order_no = 'MS12345678-90'
  AND c.customer_nm = 'Microsoft'
```

  The above query could be rewritten to make use of a view in which to encapsulate the additional joins, For example:

```
CREATE VIEW dbo.order_product
AS
SELECT
    *
FROM dbo.order AS o
  INNER JOIN dbo.product AS p
    ON  o.product_key = p.product_key
 ;

UPDATE order_line
SET dispatch_ts = CURRENT_TIMESTAMP
FROM order_product AS op
WHERE order_line.order_key = op.order_key
  AND order_line.product_key = p.product_key
  AND op.order_no = 'MS12345678-90'
  AND op.supplier_nm = 'Microsoft'
```

Migrating DELETE Statements

Although delete statement syntax can be migrated from SQL Server to PDW with few if any changes, the performance can be much better when deleting large percentages of a given large table by converting the delete statement to a CTAS statement.

Delete statements are executed on a row by row basis and are logged operations and depending on indexing complexity a lot of database pages need to be updated and freed. This can cause fragmentation as well as slow performance to delete large amounts of data due to the large amount of updates needed.

CTAS statements can provide much better performance on large deletes since the data to be preserved can be put into a new table while the old table is dropped.

In the example below the pre 1998 date would be only 20% of say a 10 billion row table or approximately 2 billion rows. In order to delete this amount of data there would be four possible options.

- **Option 1** – Run a normal delete command which would be a fairly lengthy operation.

```
DELETE FROM BigFactTable
WHERE ProductReleaseDate < 1/1/1998
```

- **Option 2** – If the table is partitioned by year or month then the appropriate partitions can be switched out to individual tables and deleted via the ALTER TABLE command syntax. This is a very fast process but depends on the table already being partitioned appropriately.

```
ALTER TABLE BigFactTable SWITCH PARTITION 2 TO
DUMMYTABLE;
DROP TABLE DUMMYTABLE;
```

- **Option 3** – Convert the delete statement to a CTAS statement by selecting the data to keep into a new table and then dropping the old table. This option is not as fast as switching out the partition but it will be many times faster than the normal delete operation and the table does not have to be pre-partitioned.

```
CREATE TABLE BigFactTable_NEW
WITH (DISTRIBUTION = HASH (ID))
AS
SELECT
    *
FROM BigFactTable
WHERE ProductReleaseDate >= 1/1/1998

RENAME OBJECT BigFactTable TO BigFactTable_OLD
RENAME OBJECT BigFactTable_NEW TO BigFactTable
```

```
DROP TABLE BigFactTable

-- The original table is distributed on a
-- column named ID.
-- The same distribution column is maintained
-- in the new table.
```

- **Option 4** – Execute a "Trickle" Delete. Which executes the delete statement from within a loop, deleting a small number of rows, approximately 100,000 each iteration. Reducing any potential logging overhead.

```
DECLARE @count INT = 1;

WHILE @count > 0
BEGIN
    IF OBJECT_ID('tempdb..#DeleteInc') IS NOT NULL
        DROP TABLE #DeleteInc

    CREATE TABLE #DeleteInc
    WITH (LOCATION=USER_DB, DISTRIBUTION=REPLICATE)
    AS
    SELECT TOP 100000
       ID
    FROM dbo.BigFactTable
    WHERE ProductReleaseDate < 1/1/1998

    SET @count = (SELECT COUNT(*) FROM #DeleteInc)

    IF @count > 0
    BEGIN
        DELETE FROM dbo.BigFactTable
        WHERE ID IN (SELECT ID FROM #DeleteInc)
    END
END
```

@@ROWCOUNT Workaround

PDW currently does not support @@ROWCOUNT or ROWCOUNT_BIG functions. If you need to obtain the number of rows affected by the last INSERT, UPDATE or DELETE statement, you can use the following SQL:

```
-- @@ROWCOUNT
SELECT CASE WHEN MAX(distribution_id) = -1
            THEN SUM(DISTINCT row_count)
            ELSE SUM(row_count)
       END AS row_count
FROM sys.dm_pdw_sql_requests
WHERE row_count <> -1
  AND request_id IN (SELECT TOP 1
                        request_id
                     FROM sys.dm_pdw_exec_requests
                     WHERE session_id = SESSION_ID()
                     ORDER BY end_time DESC)
```

Stored Procedures

Stored Procedure creation is supported in PDW and supports most features available in SQL Server. As opposed to SQL Server however the individual commands in PDW stored procedures execute in parallel across all the nodes rather than serially like they do in SQL Server.

SQL Server PDW supports nesting of Stored Procedures up to 8 levels deep compared to the 32 levels deep supported by SQL Server.

The following stored procedure features are not supported in SQL Server PDW:

- Temporary stored procedures
- Numbered stored procedures
- Table valued parameters for stored procedures
- Extended stored procedures
- CLR stored procedures
- Read-only parameters for stored procedures
- Encrypted stored procedures
- Execution of stored procedures under a different user context (the EXECUTE AS clause)
- Default parameters
- The RETURN statement

Other Functionality Limitations and Differences

The following functionality is not supported in PDW at this time.

Functions – UDF and CLR	Currently Not Supported
Rules	Rule Syntax is not supported. However defaults and constraints can be defined on tables. See the CREATE TABLE statement.
Cursors	Not supported at this time. Cursor operations are by definition designed to process data sets in a row by row fashion rather than in a set or "batch" which is the optimal way to process large amounts of data. Cursors operations are not recommended in enterprise data warehouse solutions.

Functions – UDF and CLR	Currently Not Supported
Optimizer Hints	Optimizer hints should not be migrated from SQL Server to PDW. If code is to be migrated from SQL Server query hints can be removed manually via a code review process. Optimizer Hints vary significantly between PDW and SQL Server Many of the SQL Server hints do not apply and indeed if migrated could force poor performance. The distributed execution plan that is created and executed on PDW can make hints incompatible. In PDW hints can be used to specify the join type as well as to specify if a table is to be replicated or distributed in order to resolve an incompatible query join situation. It is recommended that the use of hints is avoided unless adequate and ongoing testing verifies that hints should be used. **Statistics vs. Optimizer Hints** Up to date table statistics enable the optimizer to made the best evaluation and pick the most appropriate execution plan, Join order and join types. Conversely Invalid or old statistics can guide the optimizer to pick a sub optimal execution plan. Query hints should only be considered as a last option and only once statistics are created and up to date. Table Statistics creation is a performance optimization topic and covered in the PDW product documentation.

Security Considerations - Migration

When moving a database from SQL Server to PDW all the metadata of the dependent entities and objects in the database must be recreated on the destination platform. This includes the user security access as well as logins and passwords.

PDW supports Transparent Data Encryption (TDE), allowing for real-time I/O encryption and decryption of the data and log files. Encrypted data on SQL Server will first have to be decrypted before being able to be loaded into PDW, where if TDE has been enabled it will be encrypted and therefore protected at "rest". Certificates and Keys will need to be recreated.

Syntax Comparisons

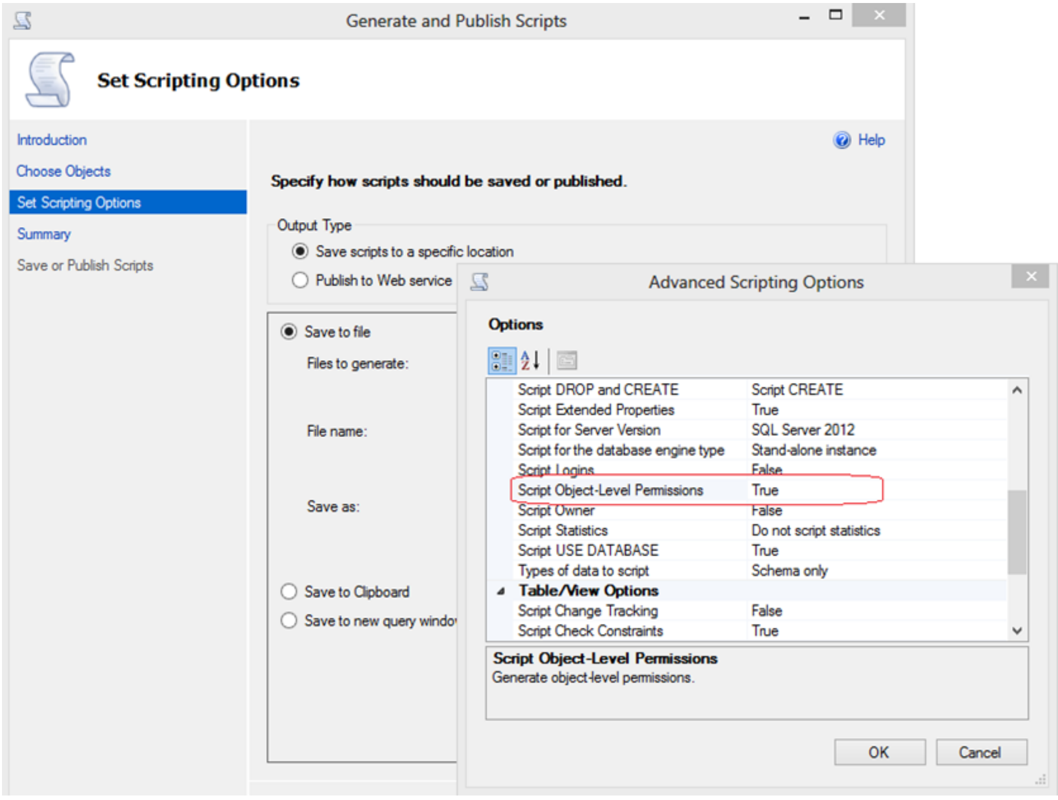
CREATE LOGIN

The CREATE LOGIN command is simplified substantially in PDW. Login creation with a certificate or an asymmetric key is not supported in the current release.

```
CREATE LOGIN loginName { WITH <option_list1> | FROM
WINDOWS }
<option_list1> ::=
    PASSWORD = { 'password' } [ MUST_CHANGE ]
    [ , <option_list2> [ ,... ] ]
<option_list2> ::=
      CHECK_EXPIRATION = { ON | OFF}
    | CHECK_POLICY = { ON | OFF}
```

When migrating any database it is important to note that while login passwords can be easily reset, user logins, user names as well as database access and permissions on objects must be maintained on the new database platform. An approach to accurately recreating the logins, users and roles on PDW is to first retrieve the metadata for these logins, users and roles on SQL Server together with any associated permissions. This can be achieved by using the database scripting wizard and specifying that logins and object level permissions should also be scripted.

The SQL Server database scripting wizard can be initiated by right clicking on a user database and selecting “Tasks” and then “Generate Scripts…” The scripting wizard will generate the SQL Statements to recreate the appropriate permissions.



The following T-SQL functions can be used to query the SQL Server security metadata, allowing you to perform command level validation on individual users and related object permissions.

NOTE: These functions are not supported by SQL Server PDW.

fn_builtin_permissions()
This function lists all of the securable classes available in SQL Server. A securable class is any object that can be assigned permissions. For example tables, views, procedures etc. The values returned will be used in the execution of the fn_my_permissions() function to list out the permissions for nominated users on nominated securable objects.

fn_my_permissions()
This lists out the permissions that a given user context has been granted on a securable object.

For example:

```
SELECT DISTINCT
    class_desc
FROM fn_builtin_permissions(default)
;
```

Lists out 25 securable objects in SQL Server 2012.

The example below lists the permissions that user "danny" has on the object "table_1". The object could be a table, view, procedure etc. Since the "fn_my_permissions()" function is executed in the current user context the user context must be set and reset as needed. The user must exist in the database.

```
EXECUTE AS USER = 'danny';
SELECT
    entity_name
  , permission_name
FROM fn_my_permissions('testdb.dbo.table_1',
'OBJECT')
ORDER BY subentity_name
       , permission_name
;
REVERT;
GO
```

This example lists the permissions that user "danny" has on the database called "testdb".

```
EXECUTE AS USER = 'danny';
SELECT
    entity_name
  , permission_name
FROM fn_my_permissions('testdb', 'DATABASE')
ORDER BY subentity_name
       , permission_name
;
REVERT;
GO
```

Conclusion

This migration guide covers the differences between SQL Server and SQL Server Parallel Data Warehouse (PDW), discussing the steps necessary to migrate a SQL Server database to SQL Server PDW.

For More Information

Analytics Platform System Website – http://www.microsoft.com/aps/

SQL Server Website - http://www.microsoft.com/sqlserver/

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

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

Feedback

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of the white papers we release.

Send feedback.

Appendix - Unsupported SQL Server Syntax

The following Syntax is not directly supported.

Category	Command	Notes
Aggregate functions	CHECKSUM_AGG	Not Applicable in PDW.
Aggregate functions	GROUPING	Not Applicable in PDW.
Aggregate functions	GROUPING_ID	Not Applicable in PDW.
ALTER	ALTER APPLICATION ROLE	Not Applicable in PDW.
ALTER	ALTER ASSEMBLY	Not Applicable in PDW.
ALTER	ALTER ASYMMETRIC KEY	Not Applicable in PDW.
ALTER	ALTER BROKER PRIORITY	Not Applicable in PDW.
ALTER	ALTER CREDENTIAL	Not Applicable in PDW.
ALTER	ALTER CRYPTOGRAPHIC PROVIDER	Not Applicable in PDW.
ALTER	ALTER DATABASE AUDIT SPECIFICATION	Not Applicable in PDW.
ALTER	ALTER DATABASE Compatibility Level	Not Applicable in PDW. Compatibility level in PDW has been set to emulate SQL Server so that third party applications continue to work.
ALTER	ALTER DATABASE Database Mirroring	Not Applicable in PDW. High availability is provided by appliance infrastructure and database mirroring is not utilized.
ALTER	ALTER DATABASE File and Filegroup Options	Not Applicable in PDW. Files and Filegroups are automatically defined by PDW and no user intervention is required.
ALTER	ALTER ENDPOINT	Not Applicable in PDW.
ALTER	ALTER EVENT SESSION	Not Applicable in PDW.
ALTER	ALTER FULLTEXT CATALOG	Not Applicable in PDW.
ALTER	ALTER FULLTEXT INDEX	Not Applicable in PDW.
ALTER	ALTER FULLTEXT STOPLIST	Not Applicable in PDW.
ALTER	ALTER FUNCTION	Not Applicable in PDW.
ALTER	ALTER MESSAGE TYPE	Not Applicable in PDW.
ALTER	ALTER PARTITION FUNCTION	Not Applicable in PDW.

Category	Command	Notes
ALTER	ALTER PARTITION SCHEME	Not Applicable in PDW.
ALTER	ALTER QUEUE	Not Applicable in PDW.
ALTER	ALTER REMOTE SERVICE BINDING	Not Applicable in PDW.
ALTER	ALTER RESOURCE GOVERNOR	Not Applicable in PDW. Resource governance is already implemented via workload management resource classes in PDW. PDW simplifies the syntax and complexity to allow resource governance during large batch execution. See books online for more information under the heading "Workload Management".
ALTER	ALTER RESOURCE POOL	Not Applicable in PDW. See books online for more information under the heading "Workload Management".
ALTER	ALTER ROUTE	Not Applicable in PDW.
ALTER	ALTER SERVER AUDIT	Not Applicable in PDW.
ALTER	ALTER SERVER AUDIT SPECIFICATION	Not Applicable in PDW.
ALTER	ALTER SERVICE	Not Applicable in PDW.
ALTER	ALTER SERVICE MASTER KEY	Not Applicable in PDW.
ALTER	ALTER SYMMETRIC KEY	Not Applicable in PDW.
ALTER	ALTER TRIGGER	Not Applicable in PDW. Triggers are not supported in PDW.
ALTER	ALTER VIEW	Not Applicable in PDW. DROP the view and re-CREATE the view instead.
ALTER	ALTER WORKLOAD GROUP	Not Applicable in PDW. See books online for more information under the heading "Workload Management".
ALTER	ALTER XML SCHEMA COLLECTION	Not Applicable in PDW. XML Data types are not supported in PDW.
BACKUP / RESTORE	BACKUP MASTER KEY	Not Applicable in PDW.
BACKUP / RESTORE	BACKUP SERVICE MASTER KEY	Use BACKUP CERTIFICATE instead.
BACKUP / RESTORE	RESTORE FILELISTONLY	Not Applicable in PDW.
BACKUP / RESTORE	RESTORE LABELONLY	Not Applicable in PDW.
BACKUP / RESTORE	RESTORE MASTER KEY	Not Applicable in PDW.
BACKUP / RESTORE	RESTORE REWINDONLY	Not Applicable in PDW.
BACKUP / RESTORE	RESTORE SERVICE MASTER KEY	Not Applicable in PDW.
BACKUP / RESTORE	RESTORE VERIFYONLY	Not Applicable in PDW.
Clause	OUTPUT Clause	Not Applicable in PDW.
Clause	WITH common_table_expression	Non-Recursive form is supported.

Category	Command	Notes
Collation	COLLATIONPROPERTY	Not Applicable in PDW.
Collation Functions / Operators	COLLATE	Not Applicable in PDW.
Collation Functions / Operators	SQL Server Collation Name	Not Applicable in PDW.
Collation Functions / Operators	Windows Collation Name	Not Applicable in PDW.
Configuration Functions	@@DBTS	Not Applicable in PDW.
Configuration Functions	@@LANGID	Not Applicable in PDW.
Configuration Functions	@@LOCK_TIMEOUT	Not Applicable in PDW.
Configuration Functions	@@MAX_CONNECTIONS	Not Applicable in PDW.
Configuration Functions	@@MAX_PRECISION	Not Applicable in PDW.
Configuration Functions	@@NESTLEVEL	Not Applicable in PDW.
Configuration Functions	@@OPTIONS	Not Applicable in PDW.
Configuration Functions	@@REMSERVER	Not Applicable in PDW.
Configuration Functions	@@SERVERNAME	Not Applicable in PDW.
Configuration Functions	@@SERVICENAME	Not Applicable in PDW.
Configuration Functions	@@TEXTSIZE	Not Applicable in PDW.
Control of Flow	CONTINUE	Not Applicable in PDW.
Control of Flow	GOTO label	Not Applicable in PDW.
Control of Flow	RETURN	Not Applicable in PDW.
Control of Flow	WAITFOR	Not Applicable in PDW.
CREATE	CREATE AGGREGATE	Not Applicable in PDW.
CREATE	CREATE APPLICATION ROLE	Not Applicable in PDW.
CREATE	CREATE ASSEMBLY	Not Applicable in PDW.
CREATE	CREATE ASYMMETRIC KEY	Not Applicable in PDW.
CREATE	CREATE BROKER PRIORITY	Not Applicable in PDW. Service Broker functionality is not supported in PDW.
CREATE	CREATE CONTRACT	Not Applicable in PDW.
CREATE	CREATE CREDENTIAL	Not Applicable in PDW.
CREATE	CREATE CRYPTOGRAPHIC PROVIDER	Not Applicable in PDW.
CREATE	CREATE DATABASE AUDIT SPECIFICATION	Not Applicable in PDW.
CREATE	CREATE DEFAULT	Not Applicable in PDW.
CREATE	CREATE ENDPOINT	Not Applicable in PDW.
CREATE	CREATE EVENT NOTIFICATION	Not Applicable in PDW.
CREATE	CREATE EVENT SESSION	Not Applicable in PDW.
CREATE	CREATE FULLTEXT CATALOG	Not Applicable in PDW.
CREATE	CREATE FULLTEXT INDEX	Not Applicable in PDW.

Category	Command	Notes
CREATE	CREATE FULLTEXT STOPLIST	Not Applicable in PDW.
CREATE	CREATE FUNCTION	Not Applicable in PDW.
CREATE	CREATE MESSAGE TYPE	Not Applicable in PDW.
CREATE	CREATE PARTITION FUNCTION	Not Applicable in PDW.
CREATE	CREATE PARTITION SCHEME	Not Applicable in PDW.
CREATE	CREATE QUEUE	Not Applicable in PDW. Service Broker functionality is not supported in PDW.
CREATE	CREATE REMOTE SERVICE BINDING	Not Applicable in PDW. Service Broker functionality is not supported in PDW.
CREATE	CREATE RESOURCE POOL	Not Applicable in PDW. See Workload management is PDW Books Online.
CREATE	CREATE ROUTE	Not Applicable in PDW.
CREATE	CREATE RULE	Not Applicable in PDW.
CREATE	CREATE SERVER AUDIT	Not Applicable in PDW.
CREATE	CREATE SERVER AUDIT SPECIFICATION	Not Applicable in PDW.
CREATE	CREATE SERVICE	Not Applicable in PDW.
CREATE	CREATE SPATIAL INDEX	Not Applicable in PDW.
CREATE	CREATE SYMMETRIC KEY	Not Applicable in PDW.
CREATE	CREATE SYNONYM	Not Applicable in PDW.
CREATE	CREATE TRIGGER	Not Applicable in PDW.
CREATE	CREATE TYPE	Not Applicable in PDW.
CREATE	CREATE WORKLOAD GROUP	Not Applicable in PDW. See books online for more information under the heading "Workload Management".
CREATE	CREATE XML INDEX	Not Applicable in PDW. XML Data types are not supported in PDW.
CREATE	CREATE XML SCHEMA COLLECTION	Not Applicable in PDW. XML Data types are not supported in PDW.
Cryptographic Functions	ASYMKEY_ID	Not Applicable in PDW.
Cryptographic Functions	ASYMKEYPROPERTY	Not Applicable in PDW.
Cryptographic Functions	CERT_ID	Not Applicable in PDW.
Cryptographic Functions	CERTPROPERTY	Not Applicable in PDW.
Cryptographic Functions	CRYPT_GEN_RANDOM	Not Applicable in PDW.
Cryptographic Functions	DECRYPTBYASMKEY	Not Applicable in PDW.
Cryptographic Functions	DECRYPTBYCERT	Not Applicable in PDW.
Cryptographic Functions	DECRYPTBYKEY	Not Applicable in PDW.
Cryptographic Functions	DECRYPTBYKEYAUTOCERT	Not Applicable in PDW.

Category	Command	Notes
Cryptographic Functions	DECRYPTBYPASSPHRASE	Not Applicable in PDW.
Cryptographic Functions	ENCRYPTBYASMKEY	Not Applicable in PDW.
Cryptographic Functions	ENCRYPTBYCERT	Not Applicable in PDW.
Cryptographic Functions	ENCRYPTBYKEY	Not Applicable in PDW.
Cryptographic Functions	ENCRYPTBYPASSPHRASE	Not Applicable in PDW.
Cryptographic Functions	IS_OBJECTSIGNED	Not Applicable in PDW.
Cryptographic Functions	KEY_GUID	Not Applicable in PDW.
Cryptographic Functions	KEY_ID	Not Applicable in PDW.
Cryptographic Functions	SIGNBYASMKEY	Not Applicable in PDW.
Cryptographic Functions	SIGNBYCERT	Not Applicable in PDW.
Cryptographic Functions	SYMKEYPROPERTY	Not Applicable in PDW.
Cryptographic Functions	VERIFYSIGNEDBYASMKEY	Not Applicable in PDW.
Cryptographic Functions	VERIFYSIGNEDBYCERT	Not Applicable in PDW.
Cursor	@@CURSOR_ROWS	Not Applicable in PDW.
Cursor	@@FETCH_STATUS	Not Applicable in PDW.
Cursor	CURSOR_STATUS	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	@@CURSOR_ROWS	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	@@FETCH_STATUS	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	CLOSE	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	CURSOR_STATUS	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	DEALLOCATE	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	DECLARE @local_variable	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	DECLARE CURSOR	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	FETCH	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	OPEN	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	SET	Not Applicable in PDW. Cursors are not supported in PDW.

Category	Command	Notes
Cursor Operations	sp_cursor_list	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	sp_describe_cursor	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	sp_describe_cursor_columns	Not Applicable in PDW. Cursors are not supported in PDW.
Cursor Operations	sp_describe_cursor_tables	Not Applicable in PDW. Cursors are not supported in PDW.
Data Type Functions	IDENT_CURRENT	Not Applicable in PDW.
Data Type Functions	IDENT_INCR	Not Applicable in PDW.
Data Type Functions	IDENT_SEED	Not Applicable in PDW.
Data Type Functions	IDENTITY	Not Applicable in PDW. Identity Fields not supported in PDW.
Data Types	cursor	Not Applicable in PDW. Cursors are not supported in PDW.
Data Types	hierarchyid	Not Supported in PDW.
Data Types	image	Not Supported in PDW.
Data Types	ntext	Not Supported in PDW.
Data Types	numeric	Not Applicable in PDW. Use DECIMAL instead
Data Types	sql_variant	Not Supported in PDW.
Data Types	table	Not Supported in PDW. Use temp tables instead.
Data Types	text	Not Supported in PDW.
Data Types	uniqueidentifier	Not Supported in PDW.
Data Types	xml	Not Supported in PDW.
DBCC	DBCC CHECKALLOC	Not Applicable in PDW.
DBCC	DBCC CHECKCATALOG	Not Applicable in PDW.
DBCC	DBCC CHECKCONSTRAINTS	Not Applicable in PDW.
DBCC	DBCC CHECKDB	Not Applicable in PDW.
DBCC	DBCC CHECKFILEGROUP	Not Applicable in PDW.
DBCC	DBCC CHECKIDENT	Not Applicable in PDW.
DBCC	DBCC CHECKTABLE	Not Applicable in PDW.
DBCC	DBCC CLEANTABLE	Not Applicable in PDW.
DBCC	DBCC DBREINDEX	Not Applicable in PDW.
DBCC	DBCC dllname (FREE)	Not Applicable in PDW.
DBCC	DBCC DROPCLEANBUFFERS	Not Applicable in PDW.

Category	Command	Notes
DBCC	DBCC FREESESSIONCACHE	Not Applicable in PDW.
DBCC	DBCC FREESYSTEMCACHE	Not Applicable in PDW.
DBCC	DBCC HELP	Not Applicable in PDW.
DBCC	DBCC INDEXDEFRAG	Not Applicable in PDW.
DBCC	DBCC INPUTBUFFER	Not Applicable in PDW.
DBCC	DBCC OPENTRAN	Not Applicable in PDW.
DBCC	DBCC OUTPUTBUFFER	Not Applicable in PDW.
DBCC	DBCC PROCCACHE	Not Applicable in PDW.
DBCC	DBCC SHOWCONTIG	Not Applicable in PDW.
DBCC	DBCC SHRINKDATABASE	Not Applicable in PDW.
DBCC	DBCC SHRINKFILE	Not Applicable in PDW.
DBCC	DBCC SQLPERF	Not Applicable in PDW.
DBCC	DBCC TRACEOFF	Not Applicable in PDW.
DBCC	DBCC TRACEON	Not Applicable in PDW.
DBCC	DBCC TRACESTATUS	Not Applicable in PDW.
DBCC	DBCC UPDATEUSAGE	Not Applicable in PDW.
DBCC	DBCC USEROPTIONS	Not Applicable in PDW.
DROP	DROP AGGREGATE	Not Applicable in PDW.
DROP	DROP APPLICATION ROLE	Not Applicable in PDW.
DROP	DROP ASSEMBLY	Not Applicable in PDW.
DROP	DROP ASYMMETRIC KEY	Not Applicable in PDW.
DROP	DROP BROKER PRIORITY	Not Applicable in PDW.
DROP	DROP CONTRACT	Not Applicable in PDW.
DROP	DROP CREDENTIAL	Not Applicable in PDW.
DROP	DROP CRYPTOGRAPHIC PROVIDER	Not Applicable in PDW.
DROP	DROP DATABASE AUDIT SPECIFICATION	Not Applicable in PDW.
DROP	DROP DEFAULT	Not Applicable in PDW.
DROP	DROP ENDPOINT	Not Applicable in PDW.
DROP	DROP EVENT NOTIFICATION	Not Applicable in PDW.
DROP	DROP EVENT SESSION	Not Applicable in PDW.
DROP	DROP FULLTEXT CATALOG	Not Applicable in PDW. Full text catalogs are not supported in PDW.
DROP	DROP FULLTEXT INDEX	Not Applicable in PDW. Full text catalogs are not supported in PDW.

Category	Command	Notes
DROP	DROP FULLTEXT STOPLIST	Not Applicable in PDW. Full text catalogs are not supported in PDW.
DROP	DROP FUNCTION	Not Applicable in PDW. Custom declared functions are not supported in PDW
DROP	DROP MESSAGE TYPE	Not Applicable in PDW.
DROP	DROP PARTITION FUNCTION	Not Applicable in PDW. Specify partition using WHERE clause.
DROP	DROP PARTITION SCHEME	Not Applicable in PDW.
DROP	DROP QUEUE	Not Applicable in PDW. Service Broker is not supported.
DROP	DROP REMOTE SERVICE BINDING	Not Applicable in PDW. Service Broker is not supported.
DROP	DROP RESOURCE POOL	Not Applicable in PDW. See Workload management is PDW Books Online.
DROP	DROP ROUTE	Not Applicable in PDW. Service Broker is not supported.
DROP	DROP RULE	Not Applicable in PDW.
DROP	DROP SERVER AUDIT	Not Applicable in PDW.
DROP	DROP SERVER AUDIT SPECIFICATION	Not Applicable in PDW.
DROP	DROP SERVICE	Not Applicable in PDW.
DROP	DROP SIGNATURE	Not Applicable in PDW.
DROP	DROP SYMMETRIC KEY	Not Applicable in PDW.
DROP	DROP SYNONYM	Not Applicable in PDW.
DROP	DROP TRIGGER	Not Applicable in PDW. Trigger functionality is not supported in PDW.
DROP	DROP TYPE	Not Applicable in PDW.
DROP	DROP WORKLOAD GROUP	Not Applicable in PDW.
DROP	DROP XML SCHEMA COLLECTION	Not Applicable in PDW.
MANAGEMENT	CHECKPOINT	Not Applicable in PDW.
MANAGEMENT	KILL QUERY NOTIFICATION SUBSCRIPTION	Not Applicable in PDW.
MANAGEMENT	KILL STATS JOB	Not Applicable in PDW.
MANAGEMENT	RECONFIGURE	Not Applicable in PDW.
MANAGEMENT	SHUTDOWN	Not Applicable in PDW.
Mathematical Functions	RAND	Not Applicable in PDW.
Metadata Functions	@@PROCID	Not Applicable in PDW.
Metadata Functions	APPLOCK_MODE	Not Applicable in PDW.
Metadata Functions	APPLOCK_TEST	Not Applicable in PDW.

Category	Command	Notes
Metadata Functions	ASSEMBLYPROPERTY	Not Applicable in PDW.
Metadata Functions	COL_LENGTH	Not Applicable in PDW.
Metadata Functions	COLUMNPROPERTY	Not Applicable in PDW.
Metadata Functions	DATABASE_PRINCIPAL_ID	Not Applicable in PDW.
Metadata Functions	DATABASEPROPERTY	Not Applicable in PDW. Use DATABASEPROPERTYEX function instead.
Metadata Functions	FILE_ID	Not Applicable in PDW.
Metadata Functions	FILE_IDEX	Not Applicable in PDW.
Metadata Functions	FILE_NAME	Not Applicable in PDW.
Metadata Functions	FILEGROUP_ID	Not Applicable in PDW.
Metadata Functions	FILEGROUP_NAME	Not Applicable in PDW.
Metadata Functions	FILEGROUPPROPERTY	Not Applicable in PDW.
Metadata Functions	FILEPROPERTY	Not Applicable in PDW.
Metadata Functions	fn_listextendedproperty	Not Applicable in PDW.
Metadata Functions	FULLTEXTCATALOGPROPERTY	Not Applicable in PDW.
Metadata Functions	FULLTEXTSERVICEPROPERTY	Not Applicable in PDW.
Metadata Functions	INDEX_COL	Not Applicable in PDW.
Metadata Functions	INDEXPROPERTY	Not Applicable in PDW.
Metadata Functions	KEY_NAME	Not Applicable in PDW.
Metadata Functions	OBJECT_DEFINITION	Not Applicable in PDW.
Metadata Functions	OBJECT_SCHEMA_NAME	Not Applicable in PDW.
Metadata Functions	ORIGINAL_DB_NAME	Not Applicable in PDW.
Metadata Functions	SCOPE_IDENTITY	Not Applicable in PDW.
Metadata Functions	STATS_DATE	Not Applicable in PDW.
ODBC Scalar Functions	OCTET_LENGTH	Partial Implementation in PDW.
ODBC Scalar Functions	TRUNCATE	Partial Implementation in PDW.
Operators (Compound)	%= (Modulo EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead
Operators (Compound)	&= (Bitwise AND EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead
Operators (Compound)	*= (Multiply EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead
Operators (Compound)	/= (Divide EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead
Operators (Compound)	^= (Bitwise Exclusive OR EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead

Category	Command	Notes
Operators (Compound)	|= (Bitwise OR EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead
Operators (Compound)	+= (Add EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead
Operators (Compound)	-= (Subtract EQUALS) (T-SQL)	Not Applicable in PDW. Use expanded syntax instead
Operators (Logical)	ALL	Not Applicable in PDW.
Operators (Logical)	ANY	Not Applicable in PDW.
Operators (Logical)	SOME	Not Applicable in PDW.
Operators (Set)	EXCEPT and INTERSECT	Not Applicable in PDW.
Operators (String Concatenation)	+= (String Concatenation)	Not Applicable in PDW.
OTHERS	BULK INSERT	Not Applicable in PDW.
OTHERS	DISABLE TRIGGER	Not Applicable in PDW. Triggers are not supported in PDW.
OTHERS	ENABLE TRIGGER	Not Applicable in PDW. Triggers are not supported in PDW.
Partition	$PARTITION	Not Applicable in PDW.
Predicates	CONTAINS	Not Applicable in PDW.
Predicates	FREETEXT	Not Applicable in PDW. Full text search is not supported in PDW
Replication	PUBLISHINGSERVERNAME	Not Applicable in PDW.
Rowset Functions	CONTAINSTABLE	Not Applicable in PDW.
Rowset Functions	FREETEXTTABLE	Not Applicable in PDW.
Rowset Functions	OPENDATASOURCE	Not Applicable in PDW.
Rowset Functions	OPENQUERY	Not Applicable in PDW.
Rowset Functions	OPENROWSET	Not Applicable in PDW.
Rowset Functions	OPENXML	Not Applicable in PDW.
SECURITY	ADD SIGNATGURE	Not Applicable in PDW.
Security	ADD SIGNATURE	Not Applicable in PDW.
SECURITY	CLOSE SYMMETRIC KEY	Not Applicable in PDW.
SECURITY	OPEN SYMMETRIC KEY	Not Applicable in PDW.
Security Functions	HAS_PERMS_BY_NAME	Not Applicable in PDW.
Security Functions	IS_MEMBER	Not Applicable in PDW.
Security Functions	IS_SRVROLEMEMBER	Not Applicable in PDW.
Security Functions	LOGINPROPERTY	Not Applicable in PDW.
Security Functions	ORIGINAL_LOGIN	Not Applicable in PDW.

Category	Command	Notes
Security Functions	PERMISSIONS	Not Applicable in PDW.
Security Functions	PWDCOMPARE	Not Applicable in PDW.
Security Functions	PWDENCRYPT	Not Applicable in PDW.
Security Functions	REVERT	Not Applicable in PDW.
Security Functions	SETUSER	Not Applicable in PDW.
Security Functions	SUSER_ID	Not Applicable in PDW.
Security Functions	SUSER_SID	Not Applicable in PDW.
Security Functions	SUSER_SNAME	Not Applicable in PDW.
Security Functions	sys.fn_builtin_permissions	Not Applicable in PDW.
Security Functions	sys.fn_my_permissions	Not Applicable in PDW.
Security Functions	USER_ID	Not Applicable in PDW.
Service Broker Statements	BEGIN CONVERSATION TIMER	Not Applicable in PDW. Service Broker is not supported
Service Broker Statements	BEGIN DIALOG CONVERSATION	Not Applicable in PDW. Service Broker is not supported
Service Broker Statements	END CONVERSATION	Not Applicable in PDW. Service Broker is not supported
Service Broker Statements	GET CONVERSATION GROUP	Not Applicable in PDW. Service Broker is not supported
Service Broker Statements	GET TRANSMISSION_STATUS	Not Applicable in PDW. Service Broker is not supported
Service Broker Statements	MOVE CONVERSATION	Not Applicable in PDW. Service Broker is not supported
Service Broker Statements	RECEIVE	Not Applicable in PDW. Service Broker is not supported
Service Broker Statements	SEND	Not Applicable in PDW. Service Broker is not supported
SET Statements	SET CURSOR_CLOSE_ON_COMMIT	Not Applicable in PDW.
SET Statements	SET DEADLOCK_PRIORITY	Not Applicable in PDW.
SET Statements	SET FIPS_FLAGGER	Not Applicable in PDW.
SET Statements	SET FORCEPLAN	Not Applicable in PDW.
SET Statements	SET IDENTITY_INSERT	Not Applicable in PDW.
SET Statements	SET NOCOUNT	Not Applicable in PDW.
SET Statements	SET NOEXEC	Not Applicable in PDW.
SET Statements	SET OFFSETS	Not Applicable in PDW.
SET Statements	SET PARSEONLY	Not Applicable in PDW.
SET Statements	SET QUERY_GOVERNOR_COST_LIMIT	Not Applicable in PDW.

Category	Command	Notes
SET Statements	SET REMOTE_PROC_TRANSACTIONS	Not Applicable in PDW.
SET Statements	SET SHOWPLAN_ALL	Not Applicable in PDW.
SET Statements	SET SHOWPLAN_TEXT	Not Applicable in PDW.
SET Statements	SET SHOWPLAN_XML	Not Applicable in PDW.
SET Statements	SET STATISTICS IO	Not Applicable in PDW.
SET Statements	SET STATISTICS PROFILE	Not Applicable in PDW.
SET Statements	SET STATISTICS TIME	Not Applicable in PDW.
SET Statements	SET STATISTICS XML	Not Applicable in PDW.
SQL Server Utilities Statements	\ (Backslash) (T-SQL)	Not Applicable in PDW.
Statements	MERGE	Not Applicable in PDW. See DWloader.exe for more information about loading data into PDW.
System Functions	@@IDENTITY	Not Applicable in PDW.
System Functions	@@ROWCOUNT	Not Applicable in PDW. A workaround has been provided in a previous section.
System Functions	APP_NAME	Not Applicable in PDW.
System Functions	BINARY_CHECKSUM	Not Applicable in PDW.
System Functions	CHECKSUM	Not Applicable in PDW.
System Functions	COLUMNS_UPDATED	Not Applicable in PDW.
System Functions	CONNECTIONPROPERTY	Not Applicable in PDW.
System Functions	CONTEXT_INFO	Not Applicable in PDW.
System Functions	CURRENT_REQUEST_ID	Not Applicable in PDW.
System Functions	ERROR_LINE	Not Applicable in PDW.
System Functions	fn_helpcollations	Not Applicable in PDW.
System Functions	fn_servershareddrives	Not Applicable in PDW.
System Functions	fn_virtualfilestats	Not Applicable in PDW.
System Functions	FORMATMESSAGE	Not Applicable in PDW.
System Functions	GET_FILESTREAM_TRANSACTION_CONTEXT	Not Applicable in PDW.
System Functions	GETANSINULL	Not Applicable in PDW.
System Functions	HOST_ID	Not Applicable in PDW.
System Functions	HOST_NAME	Not Applicable in PDW.
System Functions	MIN_ACTIVE_ROWVERSION	Not Applicable in PDW.
System Functions	NEWID	Not Applicable in PDW.
System Functions	NEWSEQUENTIALID	Not Applicable in PDW.
System Functions	PATHNAME	Not Applicable in PDW.

Category	Command	Notes
System Functions	ROWCOUNT_BIG	Not Applicable in PDW. A workaround has been provided in a previous section.
System Functions	SESSIONPROPERTY	Not Applicable in PDW.
System Functions	sys.dm_db_index_physical_stats	Not Applicable in PDW.
System Statistical Functions	@@CONNECTIONS	Not Applicable in PDW. See section in this paper relating to partitions.
System Statistical Functions	@@CPU_BUSY	Not Applicable in PDW.
System Statistical Functions	@@IDLE	Not Applicable in PDW.
System Statistical Functions	@@IO_BUSY	Not Applicable in PDW.
System Statistical Functions	@@PACK_RECEIVED	Not Applicable in PDW.
System Statistical Functions	@@PACK_SENT	Not Applicable in PDW.
System Statistical Functions	@@PACKET_ERRORS	Not Applicable in PDW.
System Statistical Functions	@@TIMETICKS	Not Applicable in PDW.
System Statistical Functions	@@TOTAL_ERRORS	Not Applicable in PDW.
System Statistical Functions	@@TOTAL_READ	Not Applicable in PDW.
System Statistical Functions	@@TOTAL_WRITE	Not Applicable in PDW.
System Statistical Functions	fn_virtualfilestats	Not Applicable in PDW.
Text & Image Functions	TEXTPTR	Not Applicable in PDW.
Text & Image Functions	TEXTVALID	Not Applicable in PDW.
Transaction Statements	BEGIN DISTRIBUTED TRANSACTION	Not Applicable in PDW.
Transaction Statements	SAVE TRANSACTION	Not Applicable in PDW.
Trigger Functions	COLUMNS_UPDATED	Not Applicable in PDW.
Trigger Functions	EVENTDATA	Not Applicable in PDW.
Trigger Functions	TRIGGER_NESTLEVEL	Not Applicable in PDW.
XML Statements	WITH XMLNAMESPACES	Not Applicable in PDW.
XML Statements	xml_schema_namespace	Not Applicable in PDW.