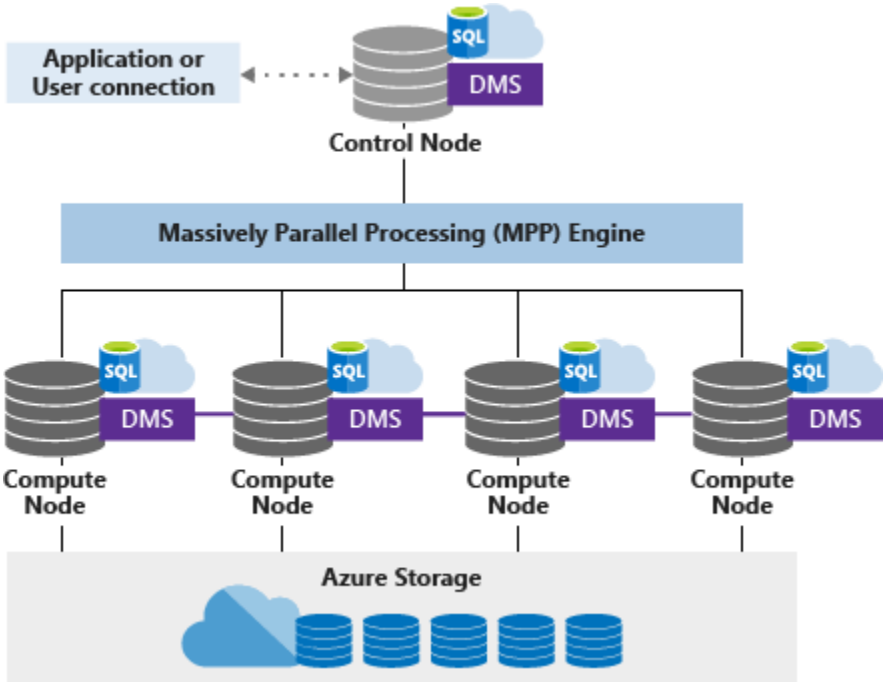
Designing distributed tables using dedicated SQL pool in Azure Synapse Analytics

Dedicated SQL pool (formerly SQL DW) architecture in Azure Synapse Analytics

Synapse SQL architecture components

Dedicated SQL pool (formerly SQL DW) leverages a scale-out architecture to distribute computational processing of data across multiple nodes. The unit of scale is an abstraction of compute power that is known as a data warehouse unit. Compute is separate from storage, which enables you to scale compute independently of the data in your system.



Dedicated SQL pool (formerly SQL DW) uses a node-based architecture. Applications connect and issue T-SQL commands to a Control node. The Control node hosts the distributed query engine, which optimizes queries for parallel processing, and then passes operations to Compute nodes to do their work in parallel.

The Compute nodes store all user data in Azure Storage and run the parallel queries. The Data Movement Service (DMS) is a system-level internal service that moves data across the nodes as necessary to run queries in parallel and return accurate results.

With decoupled storage and compute, when using a dedicated SQL pool (formerly SQL DW) one can:

- Independently size compute power irrespective of your storage needs.
- Grow or shrink compute power, within a dedicated SQL pool (formerly SQL DW), without moving data.
- Pause compute capacity while leaving data intact, so you only pay for storage.
- Resume compute capacity during operational hours.

Azure Storage

Dedicated SQL pool SQL (formerly SQL DW) leverages Azure Storage to keep your user data safe. Since your data is stored and managed by Azure Storage, there is a separate charge for your storage consumption. The data is sharded into **distributions** to optimize the performance of the system. You can choose which sharding pattern to use to distribute the data when you define the table. These sharding patterns are supported:

- Hash
- Round Robin
- Replicate

Control node

The Control node is the brain of the architecture. It is the front end that interacts with all applications and connections. The distributed query engine runs on the Control node to optimize and coordinate parallel queries. When you submit a T-SQL query, the Control node transforms it into queries that run against each distribution in parallel.

Compute nodes

The Compute nodes provide the computational power. Distributions map to Compute nodes for processing. As you pay for more compute resources, distributions are remapped to available Compute nodes. The number of compute nodes ranges from 1 to 60, and is determined by the service level for Synapse SQL.

Each Compute node has a node ID that is visible in system views. You can see the Compute node ID by looking for the node_id column in system views whose names begin with sys.pdw_nodes. For a list of these system views, see Synapse SQL system views.

Data Movement Service

Data Movement Service (DMS) is the data transport technology that coordinates data movement between the Compute nodes. Some queries require data movement to ensure the parallel queries return accurate results. When data movement is required, DMS ensures the right data gets to the right location.

Distributions

A distribution is the basic unit of storage and processing for parallel queries that run on distributed data. When Synapse SQL runs a query, the work is divided into 60 smaller queries that run in parallel.

Each of the 60 smaller queries runs on one of the data distributions. Each Compute node manages one or more of the 60 distributions. A dedicated SQL pool (formerly SQL DW) with maximum compute resources has one distribution per Compute node. A dedicated SQL pool (formerly SQL DW) with minimum compute resources has all the distributions on one compute node.

Fact tables

Fact tables are tables whose records are immutable "facts", such as service logs and measurement information. Records are progressively appended into the table in a streaming fashion or in large chunks. The records stay there until they're removed because of cost or because they've lost their value. Records are otherwise never updated.

Entity data is sometimes held in fact tables, where the entity data changes slowly. For example, data about some physical entity, such as a piece of office equipment that infrequently changes location. Since data in Kusto is immutable, the common practice is to have each table hold two columns:

- An identity (string) column that identifies the entity
- A last-modified (datetime) timestamp column

Only the last record for each entity identity is then retrieved.

Dimension tables

Dimension tables:

- Hold reference data, such as lookup tables from an entity identifier to its properties

- Hold snapshot-like data in tables whose entire contents change in a single transaction

Dimension tables aren't regularly ingested with new data. Instead, the entire data content is updated at once, using operations such as .set-or-replace, .move extents, or .rename tables.

Sometimes, dimension tables might be derived from fact tables. This process can be done via an update policy on the fact table, with a query on the table that takes the last record for each entity.

What is a distributed table?

A distributed table appears as a single table, but the rows are actually stored across 60 distributions. The rows are distributed with a hash or round-robin algorithm.

Hash-distribution improves query performance on large fact tables.**Round-robin distribution** is useful for improving loading speed. These design choices have a significant impact on improving query and loading performance.

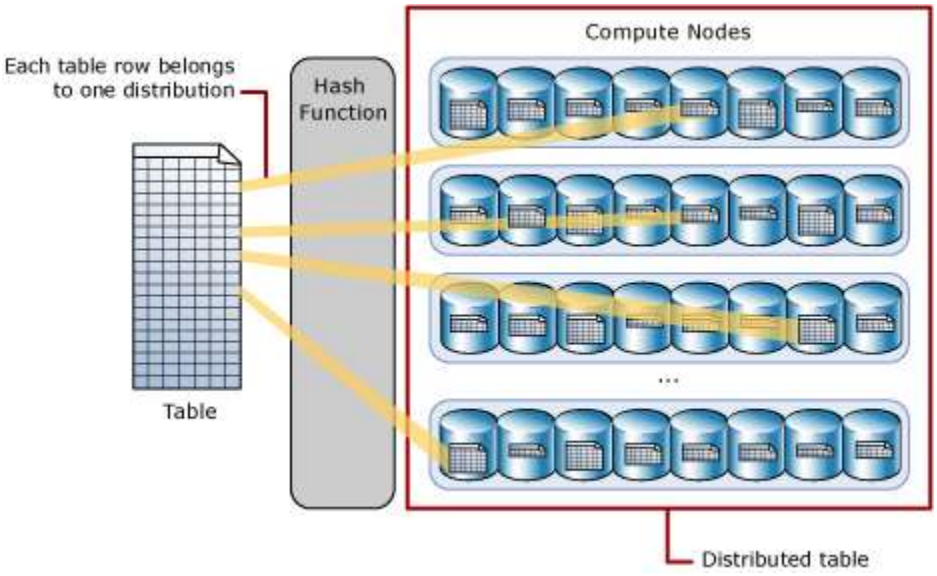
Another table storage option is to replicate a small table across all the Compute nodes. For more information, see Design guidance for replicated tables. To quickly choose among the three options, see Distributed tables in the tables overview.

As part of table design, understand as much as possible about your data and how the data is queried. For example, consider these questions:

- How large is the table?
- How often is the table refreshed?
- Do I have fact and dimension tables in a dedicated SQL pool?

Hash distributed

A hash-distributed table distributes table rows across the Compute nodes by using a deterministic hash function to assign each row to one distribution.



Since identical values always hash to the same distribution, SQL Analytics has built-in knowledge of the row locations. In dedicated SQL pool this knowledge is used to minimize data movement during queries, which improves query performance.

Hash-distributed tables work well for large fact tables in a star schema. They can have very large numbers of rows and still achieve high performance. There are some design considerations that help you to get the performance the distributed system is designed to provide. Choosing a good distribution column or columns is one such consideration that is described in this article.

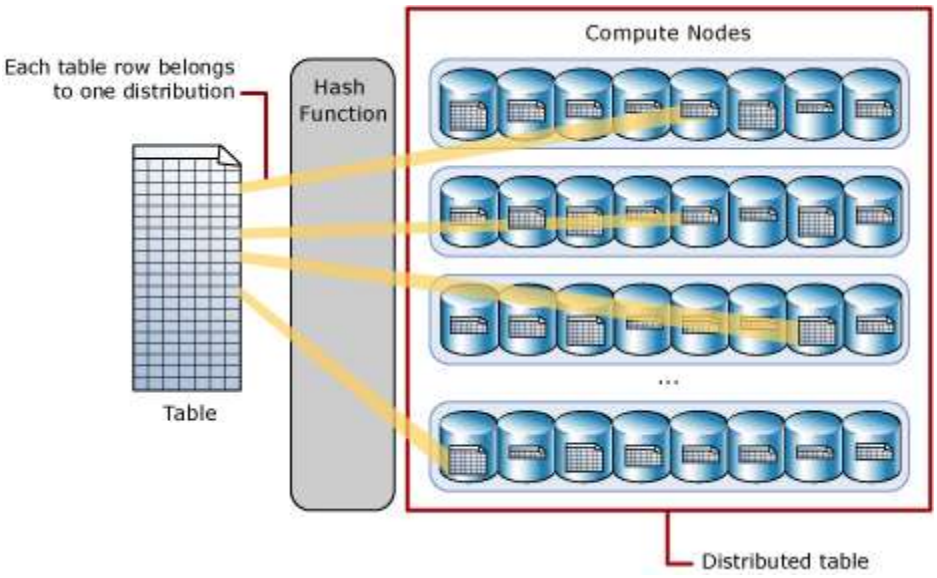
Consider using a hash-distributed table when:

- The table size on disk is more than 2 GB.
- The table has frequent insert, update, and delete operations.

A hash distributed table can deliver the highest query performance for joins and aggregations on large tables.

To shard data into a hash-distributed table, a hash function is used to deterministically assign each row to one distribution. In the table definition, one of the columns is designated as the distribution column. The hash function uses the values in the distribution column to assign each row to a distribution.

The following diagram illustrates how a full (non-distributed table) gets stored as a hash-distributed table.



- Each row belongs to one distribution.
- A deterministic hash algorithm assigns each row to one distribution.
- The number of table rows per distribution varies as shown by the different sizes of tables.

There are performance considerations for the selection of a distribution column, such as distinctness, data skew, and the types of queries that run on the system.

Round-robin distributed tables

A round-robin table is the simplest table to create and delivers fast performance when used as a staging table for loads.

A round-robin distributed table distributes data evenly across the table but without any further optimization. A distribution is first chosen at random and then buffers of rows are assigned to distributions sequentially. It is quick to load data into a round-robin table, but query performance can often be better with hash distributed tables. Joins on round-robin tables require reshuffling data, which takes additional time.

A round-robin distributed table distributes table rows evenly across all distributions. The assignment of rows to distributions is random. Unlike hash-distributed tables, rows with equal values are not guaranteed to be assigned to the same distribution.

As a result, the system sometimes needs to invoke a data movement operation to better organize your data before it can resolve a query. This extra step can slow down your queries. For example, joining a round-robin table usually requires reshuffling the rows, which is a performance hit.

Consider using the round-robin distribution for your table in the following scenarios:

- When getting started as a simple starting point since it is the default
- If there is no obvious joining key
- If there is no good candidate column for hash distributing the table
- If the table does not share a common join key with other tables
- If the join is less significant than other joins in the query
- When the table is a temporary staging table

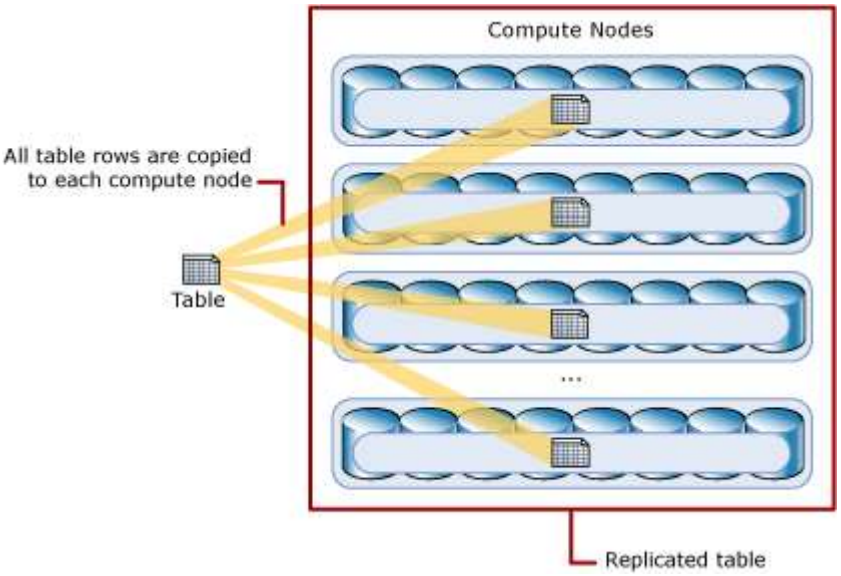
The tutorial Load New York taxicab data gives an example of loading data into a round-robin staging table.

Replicated Tables

A replicated table provides the fastest query performance for small tables.

A table that is replicated caches a full copy of the table on each compute node. Consequently, replicating a table removes the need to transfer data among compute nodes before a join or aggregation. Replicated tables are best utilized with small tables. Extra storage is required and there is additional overhead that is incurred when writing data, which make large tables impractical.

The diagram below shows a replicated table that is cached on the first distribution on each compute node.



A replicated table has a full copy of the table accessible on each Compute node. Replicating a table removes the need to transfer data among Compute nodes before a join or aggregation. Since the table has multiple copies, replicated tables work best when the table size is less than 2 GB compressed. 2 GB is not a

hard limit. If the data is static and does not change, you can replicate larger tables.

The following diagram shows a replicated table that is accessible on each Compute node. In SQL pool, the replicated table is fully copied to a distribution database on each compute node.

Replicated tables work well for dimension tables in a star schema. Dimension tables are typically joined to fact tables, which are distributed differently than the dimension table. Dimensions are usually of a size that makes it feasible to store and maintain multiple copies. Dimensions store descriptive data that changes slowly, such as customer name and address, and product details. The slowly changing nature of the data leads to less maintenance of the replicated table.

Consider using a replicated table when:

- The table size on disk is less than 2 GB, regardless of the number of rows. To find the size of a table, you can use the DBCC PDW_SHOWSPACEUSED command: DBCC PDW_SHOWSPACEUSED('ReplTableCandidate').
- The table is used in joins that would otherwise require data movement. When joining tables that are not distributed on the same column, such as a hash-distributed table to a round-robin table, data movement is required to complete the query. If one of the tables is small, consider a replicated table. We recommend using replicated tables instead of round-robin tables in most cases. To view data movement operations in query plans, use sys.dm_pdw_request_steps. The BroadcastMoveOperation is the typical data movement operation that can be eliminated by using a replicated table.

Replicated tables may not yield the best query performance when:

- The table has frequent insert, update, and delete operations. The data manipulation language (DML) operations require a rebuild of the replicated table. Rebuilding frequently can cause slower performance.
- The SQL pool is scaled frequently. Scaling a SQL pool changes the number of Compute nodes, which incurs rebuilding the replicated table.
- The table has a large number of columns, but data operations typically access only a small number of columns. In this scenario, instead of replicating the entire table, it might be more effective to distribute the table, and then create an index on the frequently accessed columns. When a query requires data movement, SQL pool only moves data for the requested columns.

Use the following strategies, depending on the table properties:

Type	**Great fit for...**	**Watch out if...**
Replicated	* Small dimension tables in a star schema with less than 2 GB of storage after compression (~5x compression)	* Many write transactions are on table (such as insert, upsert, delete, update) * You change Data Warehouse Units (DWU) provisioning frequently * You only use 2-3 columns but your table has many columns * You index a replicated table
Round Robin (default)	* Temporary/staging table * No obvious joining key or good candidate column	* Performance is slow due to data movement
Hash	* Fact tables * Large dimension tables	* The distribution key cannot be updated

Tips:

- Start with Round Robin, but aspire to a hash distribution strategy to take advantage of a massively parallel architecture.
- Make sure that common hash keys have the same data format.
- Don't distribute on varchar format.
- Dimension tables with a common hash key to a fact table with frequent join operations can be hash distributed.
- Use *sys.dm_pdw_nodes_db_partition_stats* to analyze any skewness in the data.

- Use *sys.dm_pdw_request_steps* to analyze data movements behind queries, monitor the time broadcast, and shuffle operations take. This is helpful to review your distribution strategy.

Index your table

Indexing is helpful for reading tables quickly. There is a unique set of technologies that you can use based on your needs:

Type	**Great fit for...**	**Watch out if...**
Heap	* Staging/temporary table * Small tables with small lookups	* Any lookup scans the full table
Clustered index	* Tables with up to 100 million rows * Large tables (more than 100 million rows) with only 1-2 columns heavily used	* Used on a replicated table * You have complex queries involving multiple join and Group By operations * You make updates on the indexed columns: it takes memory
Clustered columnstore index (CCI) (default)	* Large tables (more than 100 million rows)	* Used on a replicated table * You make massive update operations on your table * You overpartition your table: row groups do not span across different distribution nodes and partitions

Tips:

- On top of a clustered index, you might want to add a nonclustered index to a column heavily used for filtering.

- Be careful how you manage the memory on a table with CCI. When you load data, you want the user (or the query) to benefit from a large resource class. Make sure to avoid trimming and creating many small compressed row groups.
- On Gen2, CCI tables are cached locally on the compute nodes to maximize performance.
- For CCI, slow performance can happen due to poor compression of your row groups. If this occurs, rebuild or reorganize your CCI. You want at least 100,000 rows per compressed row groups. The ideal is 1 million rows in a row group.
- Based on the incremental load frequency and size, you want to automate when you reorganize or rebuild your indexes. Spring cleaning is always helpful.
- Be strategic when you want to trim a row group. How large are the open row groups? How much data do you expect to load in the coming days?

Partitioning

You might partition your table when you have a large fact table (greater than 1 billion rows). In 99 percent of cases, the partition key should be based on date.

With staging tables that require ELT, you can benefit from partitioning. It facilitates data lifecycle management. Be careful not to overpartition your fact or staging table, especially on a clustered columnstore index.

Incremental load

If you're going to incrementally load your data, first make sure that you allocate larger resource classes to loading your data. This is particularly important when loading into tables with clustered columnstore indexes. See resource classes for further details.

We recommend using PolyBase and ADF V2 for automating your ELT pipelines into your data warehouse.

For a large batch of updates in your historical data, consider using a CTAS to write the data you want to keep in a table rather than using INSERT, UPDATE, and DELETE.

Maintain statistics

It's important to update statistics as *significant* changes happen to your data. See update statistics to determine if *significant* changes have occurred. Updated

statistics optimize your query plans. If you find that it takes too long to maintain all of your statistics, be more selective about which columns have statistics.

You can also define the frequency of the updates. For example, you might want to update date columns, where new values might be added, on a daily basis. You gain the most benefit by having statistics on columns involved in joins, columns used in the WHERE clause, and columns found in GROUP BY.

Resource class

Resource groups are used as a way to allocate memory to queries. If you need more memory to improve query or loading speed, you should allocate higher resource classes. On the flip side, using larger resource classes impacts concurrency. You want to take that into consideration before moving all of your users to a large resource class.

If you notice that queries take too long, check that your users do not run in large resource classes. Large resource classes consume many concurrency slots. They can cause other queries to queue up.

Finally, by using Gen2 of dedicated SQL pool (formerly SQL DW), each resource class gets 2.5 times more memory than Gen1.

Lower your cost

A key feature of Azure Synapse is the ability to manage compute resources. You can pause your dedicated SQL pool (formerly SQL DW) when you're not using it, which stops the billing of compute resources. You can scale resources to meet your performance demands. To pause, use the Azure portal or PowerShell. To scale, use the Azure portal, PowerShell, T-SQL, or a REST API.

Optimize your architecture for performance

We recommend considering SQL Database and Azure Analysis Services in a hub-and-spoke architecture. This solution can provide workload isolation between different user groups while also using advanced security features from SQL Database and Azure Analysis Services. This is also a way to provide limitless concurrency to your users.

Designing hash-distributed, round-robin and replicated distributed tables in dedicated SQL pools.

Choose a distribution column

A hash-distributed table has a distribution column or set of columns that is the hash key. For example, the following code creates a hash-distributed table with ProductKey as the distribution column.

```
SQL
CREATE TABLE [dbo].[FactInternetSales]
(   [ProductKey]          int          NOT NULL
,   [OrderDateKey]         int          NOT NULL
,   [CustomerKey]          int          NOT NULL
,   [PromotionKey]         int          NOT NULL
,   [SalesOrderNumber]     nvarchar(20) NOT NULL
,   [OrderQuantity]        smallint     NOT NULL
,   [UnitPrice]            money        NOT NULL
,   [SalesAmount]          money        NOT NULL
)
WITH
(   CLUSTERED COLUMNSTORE INDEX
,  DISTRIBUTION = HASH([ProductKey])
);
```

Hash distribution can be applied on multiple columns for a more even distribution of the base table. Multi-column distribution will allow you to choose up to eight columns for distribution. This not only reduces the data skew over time but also improves query performance. For example:

```
SQL
CREATE TABLE [dbo].[FactInternetSales]
(   [ProductKey]          int          NOT NULL
,   [OrderDateKey]         int          NOT NULL
,   [CustomerKey]          int          NOT NULL
,   [PromotionKey]         int          NOT NULL
,   [SalesOrderNumber]     nvarchar(20) NOT NULL
,   [OrderQuantity]        smallint     NOT NULL
,   [UnitPrice]            money        NOT NULL
,   [SalesAmount]          money        NOT NULL
)
WITH
(   CLUSTERED COLUMNSTORE INDEX
,  DISTRIBUTION = HASH([ProductKey],   [OrderDateKey],   [CustomerKey]
,   [PromotionKey])
);
```

Note

Multi-column distribution is currently in preview for Azure Synapse Analytics. For more information on joining the preview, see multi-column distribution with **CREATE MATERIALIZED VIEW**, **CREATE TABLE**, or **CREATE TABLE AS SELECT**.

Data stored in the distribution column(s) can be updated. Updates to data in distribution column(s) could result in data shuffle operation.

Choosing distribution column(s) is an important design decision since the values in the hash column(s) determine how the rows are distributed. The best choice depends on several factors, and usually involves tradeoffs. Once a distribution column or column set is chosen, you cannot change it. If you didn't choose the best column(s) the first time, you can use CREATE TABLE AS SELECT (CTAS) to re-create the table with the desired distribution hash key.

Choose a distribution column with data that distributes evenly

For best performance, all of the distributions should have approximately the same number of rows. When one or more distributions have a disproportionate number of rows, some distributions finish their portion of a parallel query before others. Since the query can't complete until all distributions have finished processing, each query is only as fast as the slowest distribution.

- Data skew means the data is not distributed evenly across the distributions
- Processing skew means that some distributions take longer than others when running parallel queries. This can happen when the data is skewed.

To balance the parallel processing, select a distribution column or set of columns that:

- **Has many unique values.** The distribution column(s) can have duplicate values. All rows with the same value are assigned to the same distribution. Since there are 60 distributions, some distributions can have > 1 unique values while others may end with zero values.
- **Does not have NULLs, or has only a few NULLs.** For an extreme example, if all values in the distribution column(s) are NULL, all the rows are assigned to the same distribution. As a result, query processing is skewed to one distribution, and does not benefit from parallel processing.
- **Is not a date column**. All data for the same date lands in the same distribution, or will cluster records by date. If several users are all filtering

on the same date (such as today's date), then only 1 of the 60 distributions do all the processing work.

Choose a distribution column that minimizes data movement

To get the correct query result queries might move data from one Compute node to another. Data movement commonly happens when queries have joins and aggregations on distributed tables. Choosing a distribution column or column set that helps minimize data movement is one of the most important strategies for optimizing performance of your dedicated SQL pool.

To minimize data movement, select a distribution column or set of columns that:

- Is used in JOIN, GROUP BY, DISTINCT, OVER, and HAVING clauses. When two large fact tables have frequent joins, query performance improves when you distribute both tables on one of the join columns. When a table is not used in joins, consider distributing the table on a column or column set that is frequently in the GROUP BY clause.
- Is *not* used in WHERE clauses. This could narrow the query to not run on all the distributions.
- Is *not* a date column. WHERE clauses often filter by date. When this happens, all the processing could run on only a few distributions.

Once you design a hash-distributed table, the next step is to load data into the table. For loading guidance, see Loading overview.

How to tell if your distribution is a good choice

After data is loaded into a hash-distributed table, check to see how evenly the rows are distributed across the 60 distributions. The rows per distribution can vary up to 10% without a noticeable impact on performance. Consider the following topics to evaluate your distribution column(s).

Determine if the table has data skew

A quick way to check for data skew is to use DBCC PDW_SHOWSPACEUSED. The following SQL code returns the number of table rows that are stored in each of the 60 distributions. For balanced performance, the rows in your distributed table should be spread evenly across all the distributions.

SQL

```
-- Find data skew for a distributed table
```

DBCC PDW_SHOWSPACEUSED('dbo.FactInternetSales');

To identify which tables have more than 10% data skew:

1. Create the view dbo.vTableSizes that is shown in the Tables overview article.
2. Run the following query:

SQL
```
select *
from dbo.vTableSizes
where two_part_name in
    (
    select two_part_name
    from dbo.vTableSizes
    where row_count > 0
    group by two_part_name
    having (max(row_count * 1.000) - min(row_count * 1.000))/max(row_count
* 1.000) >= .10
    )
order by two_part_name, row_count;
```

Check query plans for data movement

A good distribution column set enables joins and aggregations to have minimal data movement. This affects the way joins should be written. To get minimal data movement for a join on two hash-distributed tables, one of the join columns needs to be in distribution column or column(s). When two hash-distributed tables join on a distribution column of the same data type, the join does not require data movement. Joins can use additional columns without incurring data movement.

round-robin

Creating Round-robin Tables

```
CREATE TABLE [dbo].[SalesFact](
    [ProductID] [int] NOT NULL,
    [SalesOrderID] [int] NOT NULL,
    [CustomerID] [int] NOT NULL,
    [OrderQty] [smallint]  NOT NULL,
```

```
    [UnitPrice] [money] NOT NULL,
    [OrderDate] [datetime] NULL,
    [TaxAmt] [money] NULL
)

-- To see the distribution on the table
DBCC PDW_SHOWSPACEUSED('[dbo].[SalesFact]')

-- If you execute the below query
SELECT [CustomerID], COUNT([CustomerID]) as COUNT FROM [dbo].[Sal
esFact]
GROUP BY [CustomerID]
ORDER BY [CustomerID]
```

(Or)

Load the NYC Taxi Data into SQLPOOL1

1. In Synapse Studio, navigate to the **Develop** hub, click the + button to add new resource, then create new SQL script.
2. Select the pool 'SQLPOOL1' (pool created in STEP 1 of this tutorial) in **Connect to** drop down list above the script.
3. Enter the following code:

```
SQLCopy

IF NOT EXISTS (SELECT * FROM sys.objects O JOIN sys.schemas S
ON O.schema_id = S.schema_id WHERE O.NAME =
'NYCTaxiTripSmall'  AND O.TYPE = 'U' AND S.NAME = 'dbo')
CREATE TABLE dbo.NYCTaxiTripSmall
    (
    [DateID] int,
    [MedallionID]  int,
    [HackneyLicenseID]  int,
    [PickupTimeID] int,
    [DropoffTimeID] int,
    [PickupGeographyID] int,
    [DropoffGeographyID] int,
    [PickupLatitude]  float,
    [PickupLongitude] float,
    [PickupLatLong] nvarchar(4000),
    [DropoffLatitude] float,
    [DropoffLongitude] float,
```

```
    [DropoffLatLong] nvarchar(4000),
    [PassengerCount] int,
    [TripDurationSeconds] int,
    [TripDistanceMiles]  float,
    [PaymentType] nvarchar(4000),
    [FareAmount] numeric(19,4),
    [SurchargeAmount] numeric(19,4),
    [TaxAmount] numeric(19,4),
    [TipAmount] numeric(19,4),
    [TollsAmount] numeric(19,4),
    [TotalAmount] numeric(19,4)
   )
WITH
   (
   DISTRIBUTION = ROUND_ROBIN,
    CLUSTERED COLUMNSTORE INDEX
    -- HEAP
   )
GO

COPY INTO dbo.NYCTaxiTripSmall
(DateID 1, MedallionID  2, HackneyLicenseID  3, PickupTimeID 4,
DropoffTimeID 5,
PickupGeographyID 6, DropoffGeographyID 7, PickupLatitude  8,
PickupLongitude 9,
PickupLatLong 10, DropoffLatitude 11, DropoffLongitude 12,
DropoffLatLong 13,
PassengerCount 14, TripDurationSeconds 15, TripDistanceMiles  16,
PaymentType 17,
FareAmount 18, SurchargeAmount  19, TaxAmount 20, TipAmount 21,
TollsAmount 22,
TotalAmount 23)
FROM
'https://contosolake.dfs.core.windows.net/users/NYCTripSmall.parquet'
WITH
(
   FILE_TYPE = 'PARQUET'
   ,MAXERRORS = 0
   ,IDENTITY_INSERT = 'OFF'
)
```

4. Click the **Run** button to execute the script.

5. This script will finish in less than 60 seconds. It loads 2 million rows of NYC Taxi data into a table called dbo.NYCTaxiTripSmall.

Explore the NYC Taxi data in the dedicated SQL pool

1. In Synapse Studio, go to the **Data** hub.
2. Go to **SQLPOOL1** > **Tables**.
3. Right-click the **dbo.NYCTaxiTripSmall** table and select **New SQL Script** > **Select TOP 100 Rows**.
4. Wait while a new SQL script is created and runs.
5. Notice that at the top of the SQL script **Connect to** is automatically set to the SQL pool called **SQLPOOL1**.
6. Replace the text of the SQL script with this code and run it.

   SQLCopy

   ```
   SELECT PassengerCount,
       SUM(TripDistanceMiles)  as SumTripDistance,
       AVG(TripDistanceMiles)  as AvgTripDistance
   INTO dbo.PassengerCountStats
   FROM  dbo.NYCTaxiTripSmall
   WHERE TripDistanceMiles  > 0 AND PassengerCount > 0
   GROUP BY PassengerCount
   ORDER BY PassengerCount;
   ```

   This query shows how the total trip distances and average trip distance relate to the number of passengers.

7. In the SQL script result window, change the **View** to **Chart** to see a visualization of the results as a line chart.

Replicated distributed

```
-- Lab - Creating Replicated Tables


DROP TABLE [dbo].[SalesFact]


CREATE TABLE [dbo].[SalesFact](
   [ProductID] [int] NOT NULL,
   [SalesOrderID] [int] NOT NULL,
```

```
    [CustomerID] [int] NOT NULL,
    [OrderQty] [smallint]  NOT NULL,
    [UnitPrice] [money] NOT NULL,
    [OrderDate] [datetime] NULL,
    [TaxAmt] [money] NULL
)
WITH
(
    DISTRIBUTION = REPLICATE
)

-- To see the distribution on the table
DBCC PDW_SHOWSPACEUSED('[dbo].[SalesFact]')

-- If you execute the below query
SELECT [CustomerID], COUNT([CustomerID]) as COUNT FROM [dbo].[Sal
esFact]
GROUP BY [CustomerID]
ORDER BY [CustomerID]
```