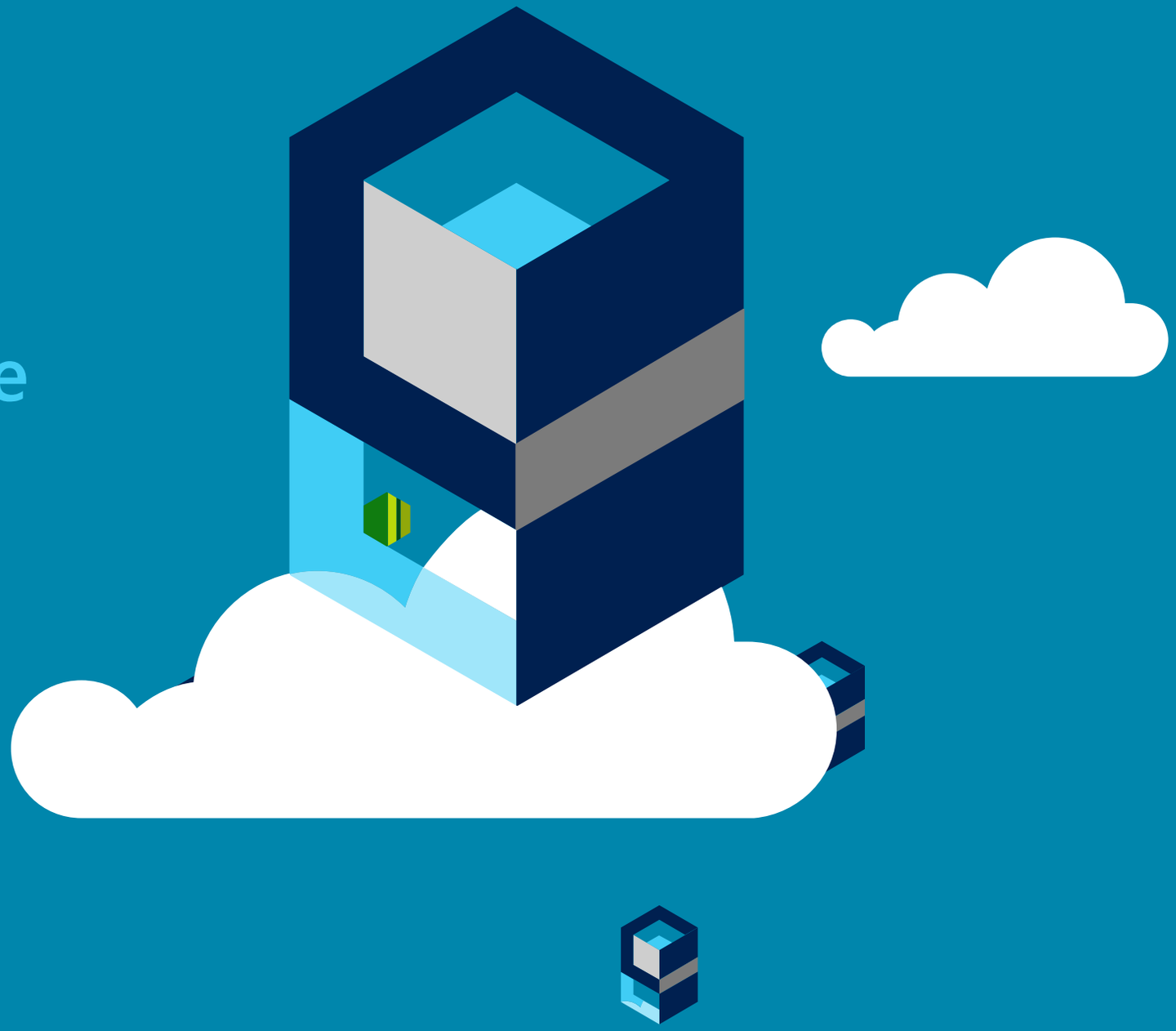
Microsoft

Modernizing Your Data Warehouse



Microsoft

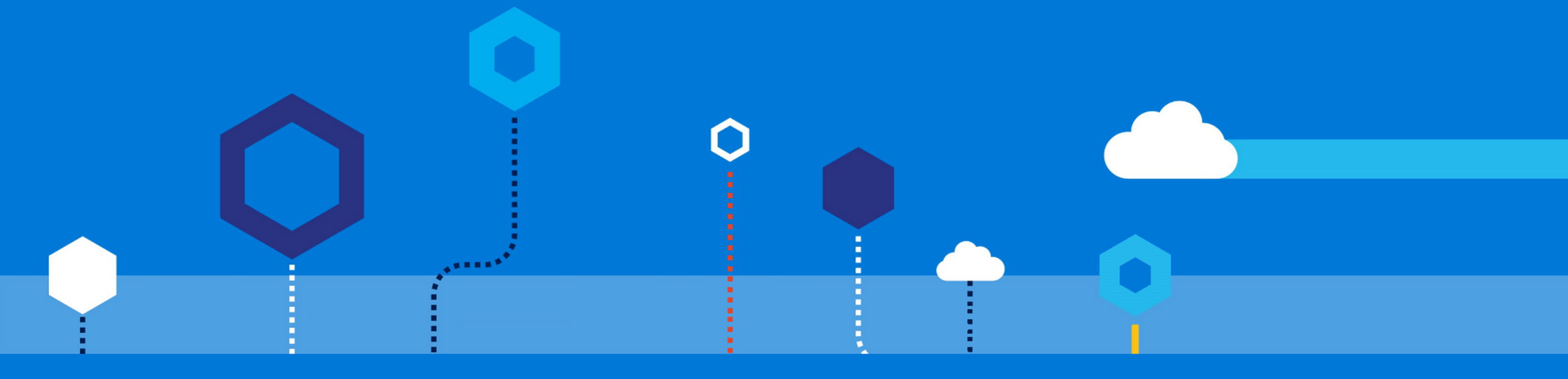
SQL DW Gen2: New Features & Enhancements

Kal Yella

Agenda

Workload Management
Interactive Query Enhancements
Security Enhancements

Workload Management

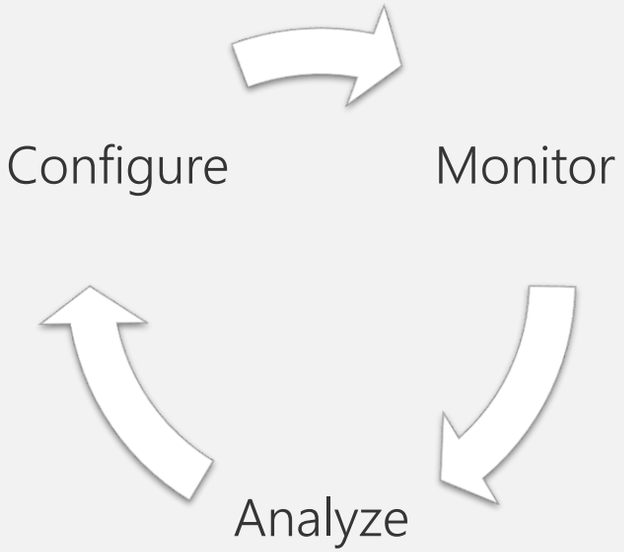


WORKLOAD MANAGEMENT LIFECYCLE

Overview

Workload management (isolation, classification, and importance) can be used dynamically in response to pressure on the data warehouse.

Configure defaults first, then monitor the pressure on the data warehouse system, and adjust configurations as necessary.

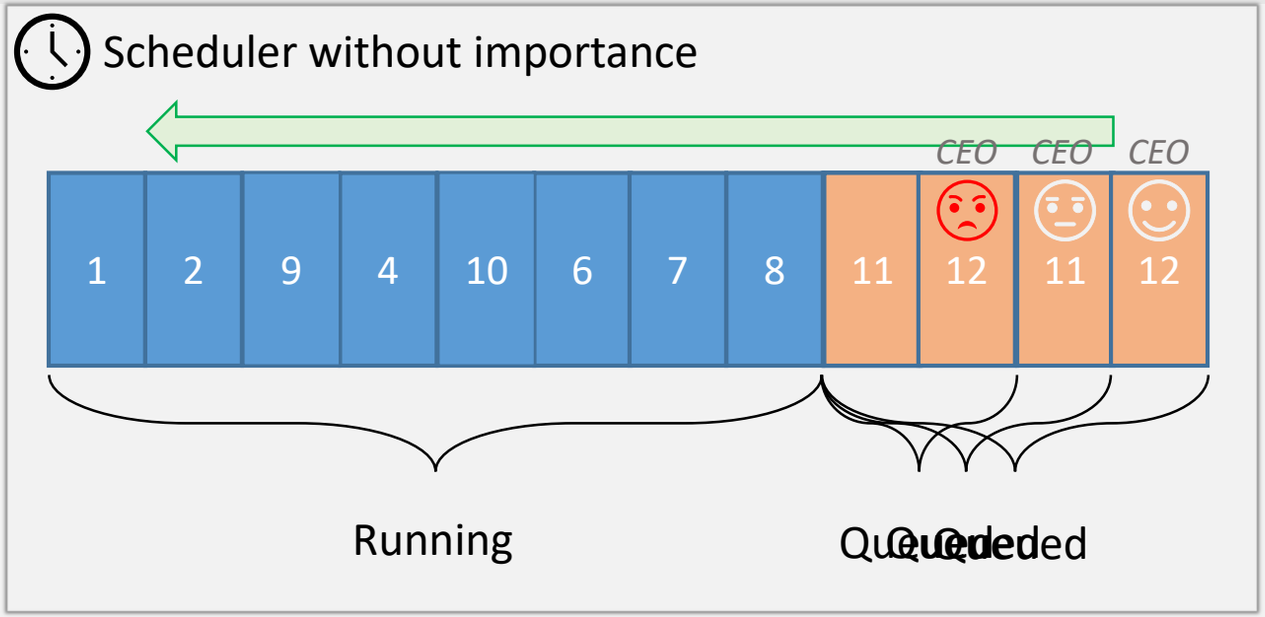


WORKLOAD IMPORTANCE – NO IMPORTANCE

Overview

Workload importance allows you to prioritize the queries that get access to resources.

It helps ensure that high-business value work is executed first on a busy data warehouse.



WORKLOAD CLASSIFICATION

Overview

Allows you to map a query to an allocation of resources via pre-determined rules

Use this in combination with workload importance to effectively share resources across different workload types

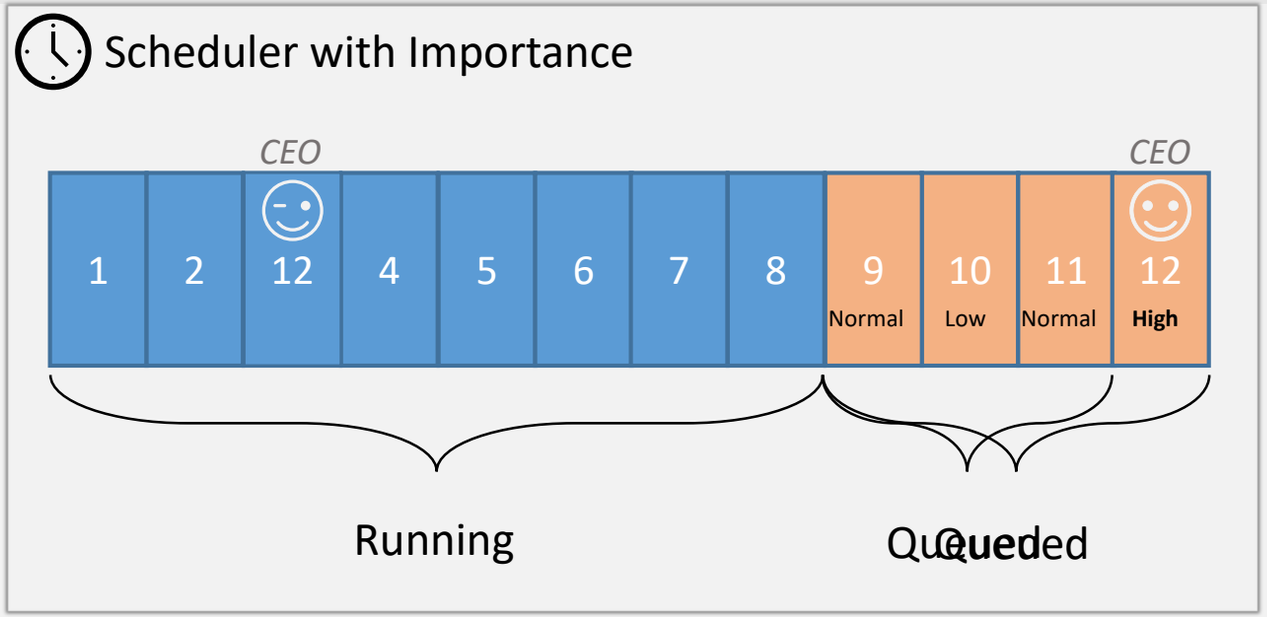
```
CREATE WORKLOAD CLASSIFIER classifier_name
WITH
    (   WORKLOAD_GROUP = 'name'
    ,   MEMBERNAME = 'security_account'
[ [ , ] WLM_LABEL = 'label' ]
[ [ , ] WLM_CONTEXT = 'context' ]
[ [ , ] START_TIME = 'HH:MM' ]
[ [ , ] END_TIME = 'HH:MM' ]

[ [ , ] IMPORTANCE = { LOW | BELOW NORMAL | NORMAL |
ABOVE NORMAL | HIGH }]) [;]
```

WORKLOAD IMPORTANCE –IMPORTANCE

Overview

Workload importance allows you to prioritize the queries that get access to resources.

It helps ensure that high-business value work is executed first on a busy data warehouse.



WORKLOAD ISOLATION (PREVIEW)

Overview

Isolation allocates fixed resources to workloads within a data warehouse. These limits are strictly enforced for memory, and only enforced under load for CPU and IO.

```
CREATE WORKLOAD GROUP group_name
 WITH
 (       MIN_PERCENTAGE_RESOURCE = value
      ,  CAP_PERCENTAGE_RESOURCE = value
      ,  REQUEST_MIN_RESOURCE_GRANT_PERCENT = value
  [ [ , ] REQUEST_MAX_RESOURCE_GRANT_PERCENT = value ]
  [ [ , ] IMPORTANCE = { LOW | BELOW NORMAL | NORMAL |
ABOVE NORMAL | HIGH }]
  [ [ , ] QUERY_EXECUTION_TIMEOUT_SEC = value ] )
  [ ; ]
```

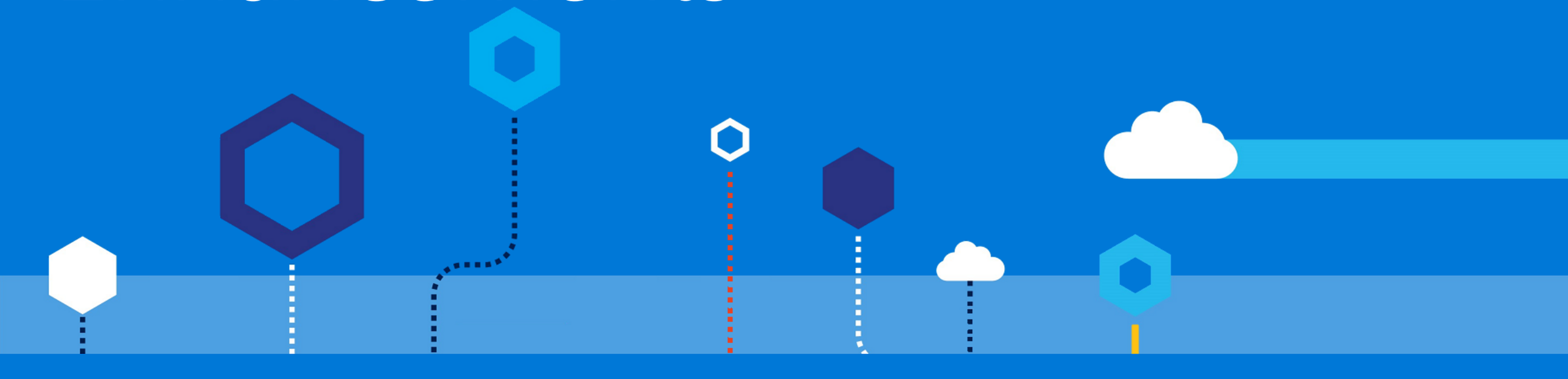
Benefits

- Reserve/Isolate/limit resources for a group of requests
- Assign policies to cancel queries that violate run times
- Assigns resources to align with processing SLAs

Workload Group with Query Timeout

```
/********* CREATE WORKLOAD GROUP *********/
CREATE WORKLOAD GROUP wgNewUsers WITH
( MIN_PERCENTAGE_RESOURCE = 0
 ,CAP_PERCENTAGE_RESOURCE = 15
 ,REQUEST_MIN_RESOURCE_GRANT_PERCENT = 3
 ,QUERY_EXECUTION_TIMEOUT_SEC = 3600 )
/********* CREATE WORKLOAD CLASSIFIER *********/
CREATE WORKLOAD CLASSIFIER wcNewUsers WITH
( WORKLOAD_GROUP = 'wgNewUsers'
 ,MEMBERNAME = 'Adhoc’ )

/********* VERIFY WORKLOAD GROUP SETTINGS *********/
SELECT [name], cap_percentage_resource, request_min_resource_grant_percent,
query_execution_timeout_sec
  FROM sys.workload_management_workload_groups
  WHERE [name] = 'wgNewUsers'
```

Interactive Query Enhancements



Result-set caching

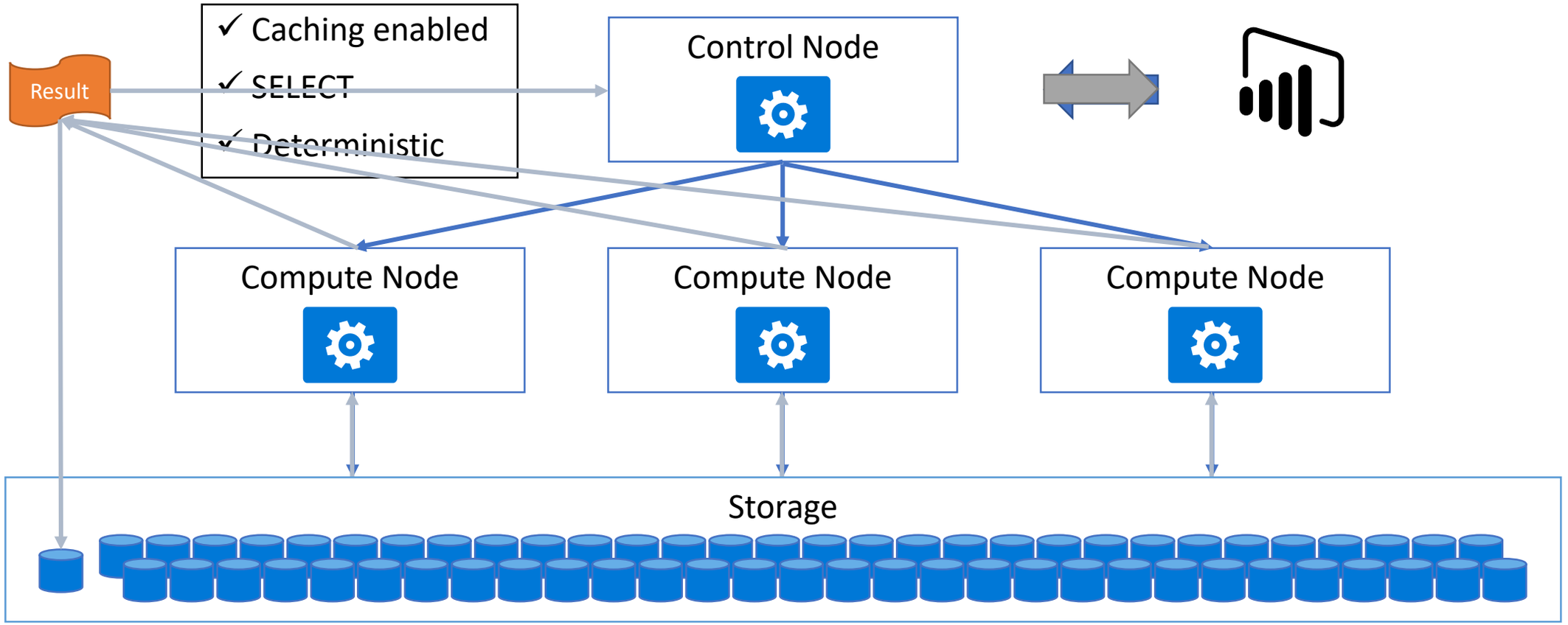
Overview

- Cache the results of a query in DW storage.
- Enables interactive response times for repetitive queries
- Ideal for tables with infrequent data changes.
- Persists even if a data warehouse is paused and resumed later.
- Invalidated and refreshed when underlying table data or query code changes.
- Evicted regularly based on a time-aware least recently used algorithm (TLRU).

```
-- Turn on/off result-set caching for a database
-- Must be run on the MASTER database
ALTER DATABASE {database_name}
SET RESULT_SET_CACHING { ON | OFF }

-- Turn on/off result-set caching for a client
session
-- Run on target data warehouse
SET RESULT_SET_CACHING {ON | OFF}

-- Check result-set caching setting for a database
-- Run on target data warehouse
SELECT is_result_set_caching_on
FROM   sys.databases
WHERE  name = {database_name}

-- Return all query requests with cache hits
SELECT *
FROM   sys.dm_pdw_request_steps
WHERE  command like '%DWResultCacheDb%'
       AND step_index = 0

SELECT result_cache_hit
FROM   sys.dm_pdw_exec_requests
```

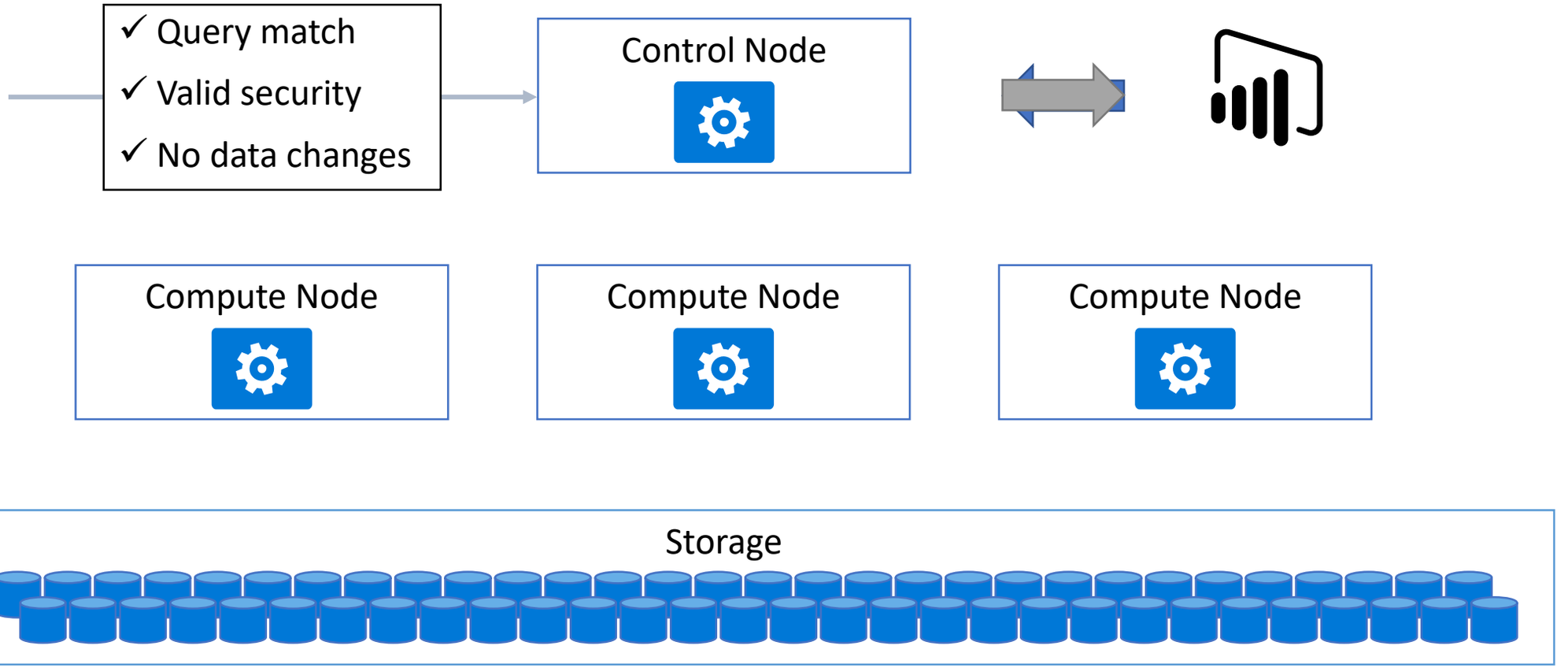
Run animation

Initial Query Execution with Result Caching



Run animation

Repetitive Execution with Result Caching



Caching Considerations

- Top level SELECT queries only (no sub-query)
- Result needs to be deterministic
  - No runtime constants (For example, GETDATE(), CURRENT_USER()…)
  - No row level security predicates on any of table reference
- No temp/system/external table reference
- Cached Results will be retrieved if:
  - The user executing the query has access to all the tables referenced in the query.
  - There is an exact match between the new query and the previous query that generated the result set cache.
  - There is no data or schema changes in the tables where the cached result set was generated from.

Indexed (materialized) views

Overview

Caches the schema and data for a view in DW remote storage.

Useful for improving the performance of 'SELECT' statement queries that include aggregations

Synchronously updated when data in underlying tables are changed.

The auto caching functionality allows SQL DW Query Optimizer to consider using indexed view even if the view is not referenced in the query.

Supported aggregations: MAX, MIN, AVG, COUNT, COUNT_BIG, SUM, VAR, STDEV

Materialized View recommendation via EXPLAIN WITH_RECOMMENDATIONS command

```
-- Create indexed view
CREATE INDEXED VIEW Sales.vw_Orders
WITH
(
   DISTRIBUTION = ROUND_ROBIN |
   HASH(ProductID)
)
AS
    SELECT SUM(UnitPrice*OrderQty) AS Revenue,
           OrderDate,
           ProductID,
           COUNT_BIG(*) AS OrderCount
    FROM   Sales.SalesOrderDetail
    GROUP  BY OrderDate, ProductID;
GO

-- Disable index view and put it in suspended mode
ALTER INDEX ALL ON Sales.vw_Orders DISABLE;

-- Re-enable index view by rebuilding it
ALTER INDEX ALL ON Sales.vw_Orders REBUILD;

--To find all materialized views in a database:
SELECT V.name, V.object_id, I.type_desc
FROM sys.views V
JOIN sys.indexes I ON V.object_id= I.object_id and
I.index_id < 2
```

Indexed (materialized) views - example

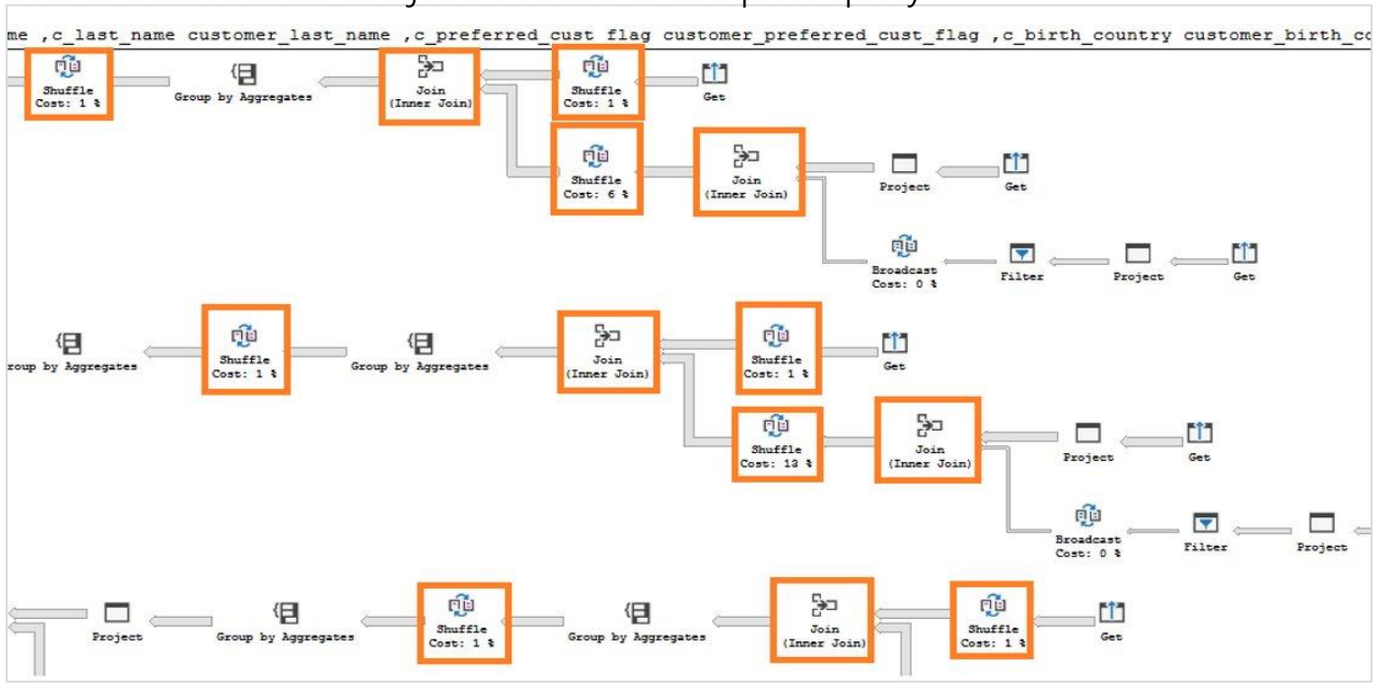
In this example, a query to get the year total sales per customer is shown to have a lot of data shuffles and joins that contribute to slow performance:

No relevant indexed views created on the data warehouse

```
-- Get year total sales per customer
(WITH year_total AS
     SELECT customer_id,
            first_name,
            last_name,
            birth_country,
            login,
            email_address,
            d_year,
            SUM(ISNULL(list_price - wholesale_cost -
            discount_amt + sales_price, 0)/2)year_total
     FROM   customer cust
     JOIN   catalog_sales sales ON cust.sk = sales.sk
     JOIN   date_dim ON sales.sold_date = date_dim.date
     GROUP  BY customer_id, first_name,
           last_name,birth_country,
           login,email_address ,d_year
)
SELECT TOP 100 …
FROM   year_total …
WHERE  …
ORDER BY …
```

Execution time: 103 seconds

Lots of data shuffles and joins needed to complete query



Indexed (materialized) views - example

Now, we add an indexed view to the data warehouse to increase the performance of the previous query. This view can be leveraged by the query even though it is not directly referenced.

Original query – get year total sales per customer

```
-- Get year total sales per customer
(WITH year_total AS
     SELECT customer_id,
            first_name,
            last_name,
            birth_country,
            login,
            email_address,
            d_year,
            SUM(ISNULL(list_price - wholesale_cost -
            discount_amt + sales_price, 0)/2)year_total
     FROM   customer cust
     JOIN   catalog_sales sales ON cust.sk = sales.sk
     JOIN   date_dim ON sales.sold_date = date_dim.date
     GROUP  BY customer_id, first_name,
           last_name,birth_country,
           login,email_address ,d_year
)
SELECT TOP 100 …
FROM   year_total …
WHERE  …
ORDER BY …
```

Create indexed view with hash distribution on customer_id column

```
-- Create indexed view for query
CREATE INDEXED VIEW nbViewCS WITH (DISTRIBUTION=HASH(customer_id)) AS
SELECT customer_id,
       first_name,
       last_name,
       birth_country,
       login,
       email_address,
       d_year,
       SUM(ISNULL(list_price - wholesale_cost - discount_amt +
       sales_price, 0)/2) AS year_total
FROM   customer cust
JOIN   catalog_sales sales ON cust.sk = sales.sk
JOIN   date_dim ON sales.sold_date = date_dim.date
GROUP  BY customer_id, first_name,
       last_name,birth_country,
       login, email_address, d_year
```

Indexed (materialized) views - example

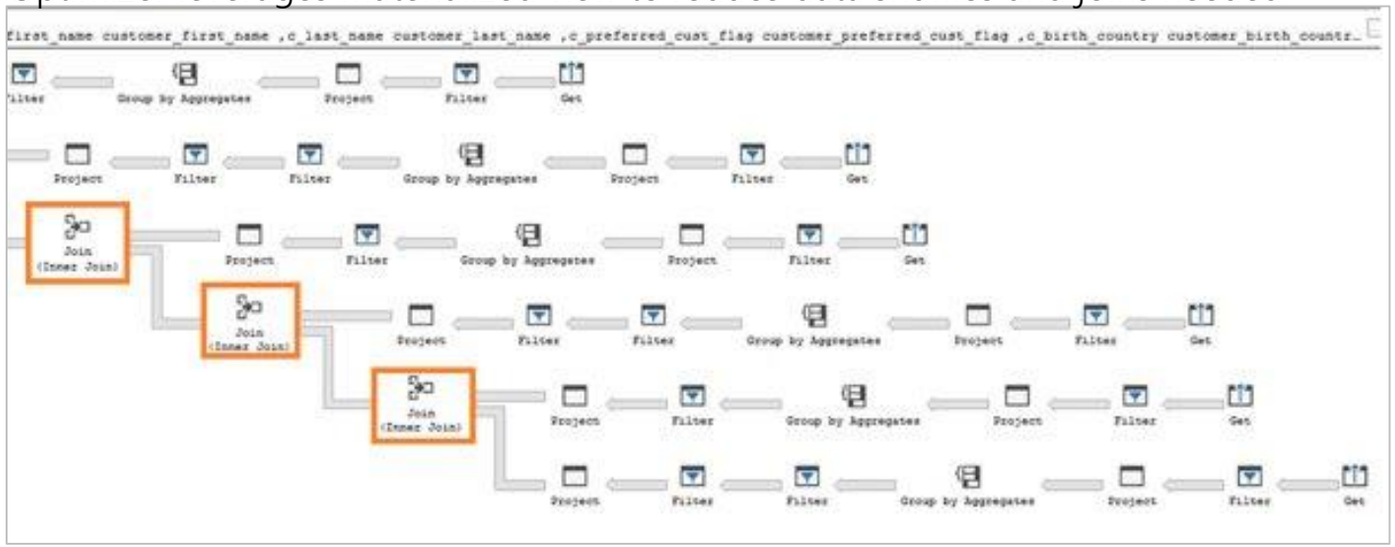
The SQL Data Warehouse query optimizer automatically leverages the indexed view to speed up the same query.
Notice that the query does not need to reference the view directly

Original query – no changes have been made to query

```
-- Get year total sales per customer
(WITH year_total AS
     SELECT customer_id,
            first_name,
            last_name,
            birth_country,
            login,
            email_address,
            d_year,
            SUM(ISNULL(list_price - wholesale_cost -
            discount_amt + sales_price, 0)/2)year_total
     FROM   customer cust
     JOIN   catalog_sales sales ON cust.sk = sales.sk
     JOIN   date_dim ON sales.sold_date = date_dim.date
     GROUP  BY customer_id, first_name,
           last_name,birth_country,
           login,email_address ,d_year
)
SELECT TOP 100 …
FROM   year_total …
WHERE  …
ORDER BY …
```

Execution time: 6 seconds

Optimizer leverages materialized view to reduce data shuffles and joins needed



Materialized View Considerations

The SELECT list in the materialized view definition needs to meet at least one of these two criteria:

The SELECT list contains an aggregate function.

GROUP BY is used in the Materialized view definition and all columns in GROUP BY are included in the SELECT list.

Materialized views can be created on partitioned tables.

SPLIT / MERGE are supported on tables referenced in materialized view.

SWITCH is not supported on tables referenced in materialized view.

A materialized view is stored in DW just like a table with clustered columnstore index (CCI)

Run SP_SPACEUSED and DBCC PDW_SHOWSPACEUSED to check space used by materialized view.

Best Practices

Avoid having many materialized views in one database with overlapping base tables.

MV recommendations returned by EXPLAIN for one query should be evaluated with your workload needs.

Drop materialized view with low usage.

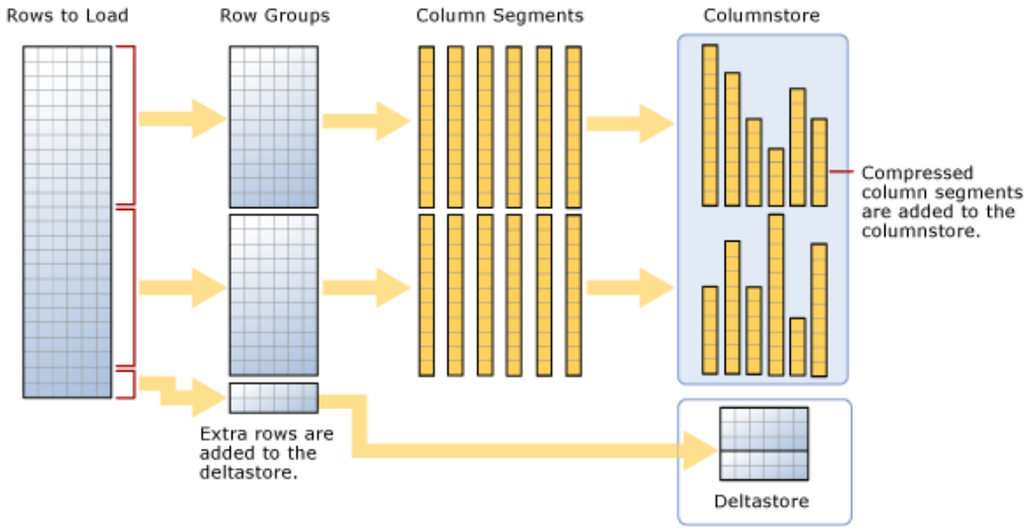
Monitor materialized view overhead and rebuild materialized view to eliminate rows in delta store.

Ordered Columnstore Segments

Overview

Queries against tables with ordered columnstore segments can take advantage of improved segment elimination to drastically reduce the time needed to service a query.

Columnstore Segments are automatically updated as data is inserted, updated, or deleted in data warehouse tables.




```
-- Create Table with Ordered Columnstore Index
CREATE TABLE sortedOrderTable
(
    OrderId  INT NOT NULL,
    Date     DATE NOT NULL,
    Name     VARCHAR(2),
    Country  VARCHAR(2)
)
WITH
(
  CLUSTERED COLUMNSTORE INDEX ORDER (OrderId)
)
-- Create Clustered Columnstore Index on existing table
CREATE CLUSTERED COLUMNSTORE INDEX cciOrderId
ON dbo.OrderTable ORDER (OrderId)

-- Insert data into table with ordered columnstore
index
INSERT INTO sortedOrderTable
VALUES (1, '01-01-2019','Dave', 'UK')
```

Snapshot isolation

Overview

Specifies that statements cannot read data that has been modified but not committed by other transactions.

This prevents dirty reads.

Isolation level

READ_COMMITTED_SNAPSHOT

OFF (Default) – Uses shared locks to prevent other transactions from modifying rows while running a read operation

ON – Uses row versioning to present each statement with a transactionally consistent snapshot of the data as it existed at the start of the statement. Locks are not used to protect the data from updates.

```
ALTER DATABASE MyDatabase
SET READ_COMMITTED_SNAPSHOT ON
```

Loading Data – Simplified

- **Simplified syntax**
  - COPY INTO <table>
- **Simplified security permissions**
  - No CONTROL permissions required
  - GRANT SELECT
  - GRANT INSERT
  - GRANT ADMINISTER BULK OPERATIONS
  - Shared Access Signature Support
- Improvements
  - **CSV support with escape delimiters**
  - **Custom row terminators**
  - **..and more!**

VS

1
```
CREATE MASTER KEY;
```

2
```
CREATE DATABASE SCOPED CREDENTIAL <credname>
WITH
     IDENTITY = '',
     SECRET = 'AZURE STORAGE ACCOUNT KEY>';
```

3
```
CREATE EXTERNAL DATA SOURCE <DSName>
WITH (TYPE=HADOOP, LOCATION=…, CREDENTIAL=…);
```

4
```
CREATE EXTERNAL FILE FORMAT <fileformat>
WITH (FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS ( FIELD_TERMINATOR = '|',…) );
```

5
```
CREATE EXTERNAL TABLE [ext].<tablename>
(
  <Column definitions>
) WITH ( LOCATION = '<location>'
, DATA_SOURCE = <DSName>
, …);
```

6
```
CREATE TABLE [dbo].[staging_table]
WITH
(
    DISTRIBUTION = ROUND_ROBIN,CLUSTERED COLUMNSTORE INDEX
)
AS
SELECT * FROM    [ext].<tablename>;
```

1
```
COPY INTO dbo.[lineitem] FROM
'https://unsecureaccount.blob.core.windows.net/customerdatasets/linitem.csv'
```

COPY vs PolyBase

Functionality	Copy Command	Polybase i.e. external table
Simplified load	Yes	No [needs minimum 6 objects to load data]
Escape all delimiters	Yes [Row, Field, and String]	Field only
Custom row terminator	Yes	No
Access permissions	Simplified [Administer Bulk Operations]	Highly privileged [Control permissions]
Custom default value for each column	Yes	No
SAS Key	Yes	No
Different storage accounts for error file and data file	Yes	No
SQL Server date format for CSV file	Yes	No
Load data even when number of columns in the source file and destination table differ	Yes	No
Rows up to 2GB in size for CSV files	2GB	1MB

JSON data support – insert JSON data

Overview

The JSON format enables representation of complex or hierarchical data structures in tables.

JSON data is stored using standard NVARCHAR table columns.

```
-- Create Table with column for JSON string
CREATE TABLE CustomerOrders
(
  CustomerId   BIGINT NOT NULL,
  Country      NVARCHAR(150) NOT NULL,
  OrderDetails NVARCHAR(3000) NOT NULL -- NVARCHAR column for JSON
) WITH (DISTRIBUTION = ROUND_ROBIN)

-- Populate table with semi-structured data
INSERT INTO CustomerOrders
VALUES
( 101, -- CustomerId
  'Bahrain', -- Country
  N'[{ StoreId": "AW73565",
       "Order": { "Number":"SO43659",
                  "Date":"2011-05-31T00:00:00"
                },
       "Item":  { "Price":2024.40, "Quantity":1 }
     }]' -- OrderDetails
)
```

JSON data support – read JSON data

Overview

Read JSON data stored in a string column with the following:

- ISJSON – verify if text is valid JSON
- JSON_VALUE – extract a scalar value from a JSON string
- JSON_QUERY – extract a JSON object or array from a JSON string

```
-- Return all rows with valid JSON data
SELECT CustomerId, OrderDetails
FROM   CustomerOrders
WHERE  ISJSON(OrderDetails) > 0;
```

CustomerId	OrderDetails
101	N'[{ StoreId": "AW73565", "Order": { "Number":"SO43659", "Date":"2011-05-31T00:00:00" }, "Item": { "Price":2024.40, "Quantity":1 }}]'

```
-- Extract values from JSON string
SELECT CustomerId,
       Country,
       JSON_VALUE(OrderDetails,'$.StoreId') AS StoreId,
       JSON_QUERY(OrderDetails,'$.Item') AS ItemDetails
FROM   CustomerOrders;
```

CustomerId	Country	StoreId	ItemDetails
101	Bahrain	AW73565	{ "Price":2024.40, "Quantity":1 }

JSON data support – modify and operate on JSON data

Overview

Use standard table columns and values from JSON text in the same analytical query.

Modify JSON data with the following:

- JSON_MODIFY – modifies a value in a JSON string
- OPENJSON – convert JSON collection to a set of rows and columns

```
-- Modify Item Quantity value
UPDATE CustomerOrders SET OrderDetails =
JSON_MODIFY(OrderDetails, '$.OrderDetails.Item.Quantity',2)
```

OrderDetails
N'[{ StoreId": "AW73565", "Order": { "Number":"SO43659", "Date":"2011-05-31T00:00:00" }, "Item": { "Price":2024.40, "Quantity": 2}}]'

```
-- Convert JSON collection to rows and columns
SELECT CustomerId,
       StoreId,
       OrderDetails.OrderDate,
       OrderDetails.OrderPrice
FROM   CustomerOrders
CROSS APPLY OPENJSON (CustomerOrders.OrderDetails)
WITH ( StoreId       VARCHAR(50)  '$.StoreId',
       OrderNumber   VARCHAR(100) '$.Order.Date',
       OrderDate     DATETIME     '$.Order.Date',
       OrderPrice    DECIMAL      '$.Item.Price',
       OrderQuantity INT          '$.Item.Quantity'
     ) AS OrderDetails
```

CustomerId	StoreId	OrderDate	OrderPrice
101	AW73565	2011-05-31T00:00:00	2024.40

Windowing functions

OVER clause

Defines a window or specified set of rows within a query result set

Computes a value for each row in the window

Aggregate functions

COUNT, MAX, AVG, SUM, APPROX_COUNT_DISTINCT, MIN, STDEV, STDEVP, STRING_AGG, VAR, VARP, GROUPING, GROUPING_ID, COUNT_BIG, CHECKSUM_AGG

Analytical functions

LAG, LEAD, FIRST_VALUE, LAST_VALUE, CUME_DIST, PERCENTILE_CONT, PERCENTILE_DISC, PERCENT_RANK

Ranking functions

RANK, NTILE, DENSE_RANK, ROW_NUMBER

ROWS | RANGE

PRECEDING, UNBOUNDING PRECEDING, CURRENT ROW, BETWEEN, FOLLOWING, UNBOUNDED FOLLOWING

```
SELECT
   ROW_NUMBER() OVER(PARTITION BY PostalCode ORDER BY SalesYTD DESC
) AS "Row Number",
   LastName,
   SalesYTD,
   PostalCode
FROM Sales
WHERE SalesYTD <> 0
ORDER BY PostalCode;
```

Row Number	LastName	SalesYTD	PostalCode
1	Mitchell	4251368.5497	98027
2	Blythe	3763178.1787	98027
3	Carson	3189418.3662	98027
4	Reiter	2315185.611	98027
5	Vargas	1453719.4653	98027
6	Ansman-Wolfe	1352577.1325	98027
1	Pak	4116870.2277	98055
2	Varkey Chudukaktil	3121616.3202	98055
3	Saraiva	2604540.7172	98055
4	Ito	2458535.6169	98055
5	Valdez	1827066.7118	98055
6	Mensa-Annan	1576562.1966	98055
7	Campbell	1573012.9383	98055
8	Tsoflias	1421810.9242	98055

Group by options

Group by with rollup

Creates a group for each combination of column expressions.

Rolls up the results into subtotals and grand totals.

Grouping sets

Combine multiple GROUP BY clauses into one GROUP BY CLAUSE.

Equivalent of UNION ALL of specified groups.

```
-- GROUP BY SETS Example --
SELECT Country,
SUM(Sales) AS TotalSales
FROM Sales
GROUP BY GROUPING SETS ( Country, () );
```

```
-- GROUP BY ROLLUP Example --
SELECT Country,
Region,
SUM(Sales) AS TotalSales
FROM Sales
GROUP BY ROLLUP (Country, Region);
-- Results --
```

Country	Region	TotalSales
Canada	Alberta	100
Canada	British Columbia	500
Canada	NULL	600
United States	Montana	100
United States	NULL	100
NULL	NULL	700

DATABRICKS – STRUCTURED STREAMING

Overview

The Databricks SQL DW connector supports batch and structured streaming support for writing real-time data into Azure SQL Data Warehouse.

It uses Polybase and the Databricks structured streaming API to stream data from Kafka, Kinesis sources directly into SQL DW at a user-configurable rate.

Source: https://docs.azuredatabricks.net/spark/latest/data-sources/azure/sql-data-warehouse.html#streaming-support

```
# Prepare streaming source; this could be Kafka,
Kinesis, or a simple rate stream.
df = spark.readStream \
  .format("rate") \
  .option("rowsPerSecond", "100000") \
  .option("numPartitions", "16") \
  .load()

# Apply some transformations to the data then use
# Structured Streaming API to continuously write the
data to a table in SQL DW.
df.writeStream \
  .format("com.databricks.spark.sqldw") \
  .option("url", <azure-sqldw-jdbc-url>) \
  .option("tempDir",
"wasbs://<containername>@<storageaccount>.blob.core.
windows.net/<directory>") \
  .option("forwardSparkAzureStorageCredentials",
"true") \
  .option("dbTable", <table-name>) \
  .option("checkpointLocation", "/tmp_location") \
  .start()
```

Microsoft

Q&A

Microsoft

Modernizing Your Data Warehouse