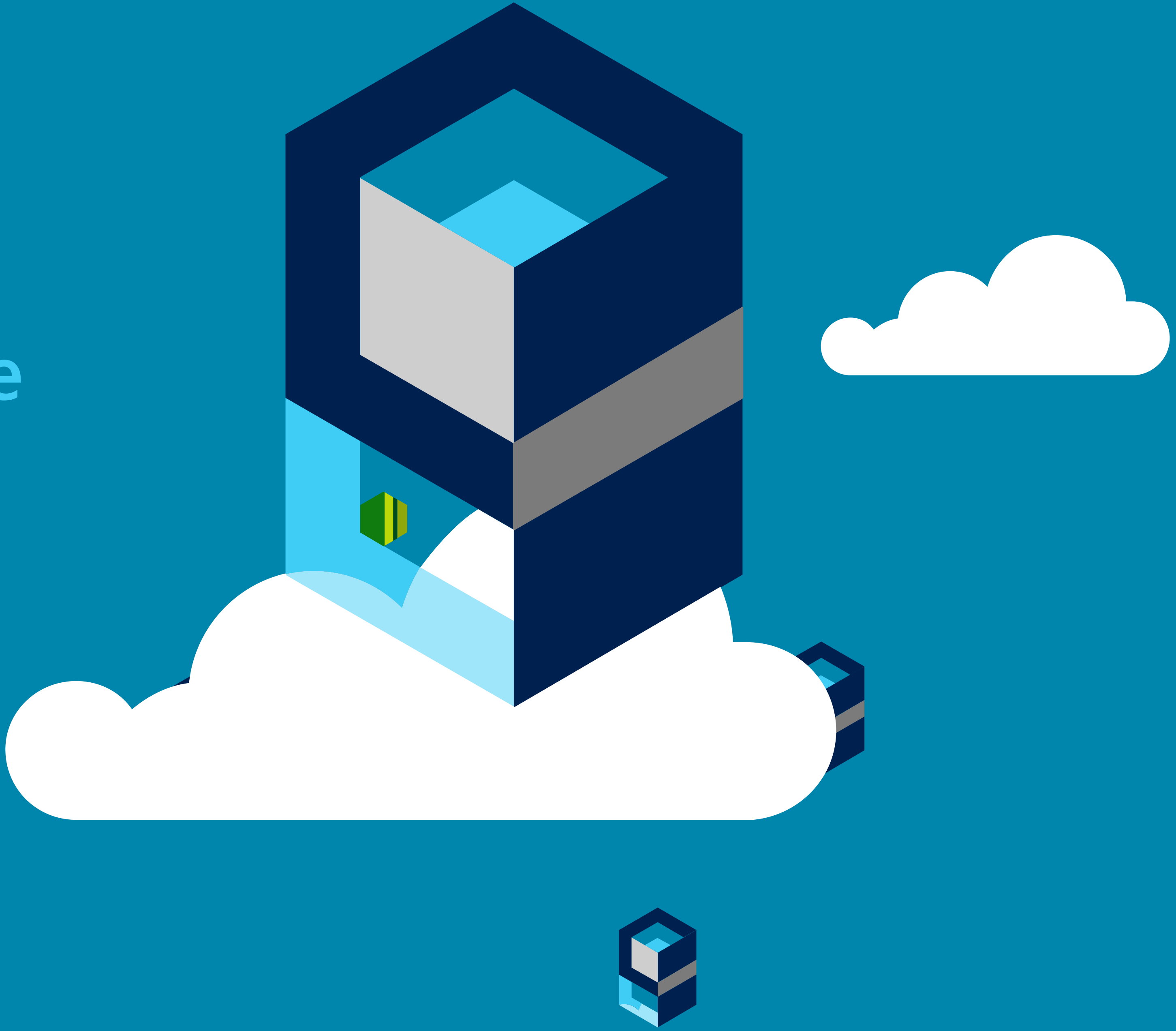
Microsoft

Modernizing Your Data Warehouse

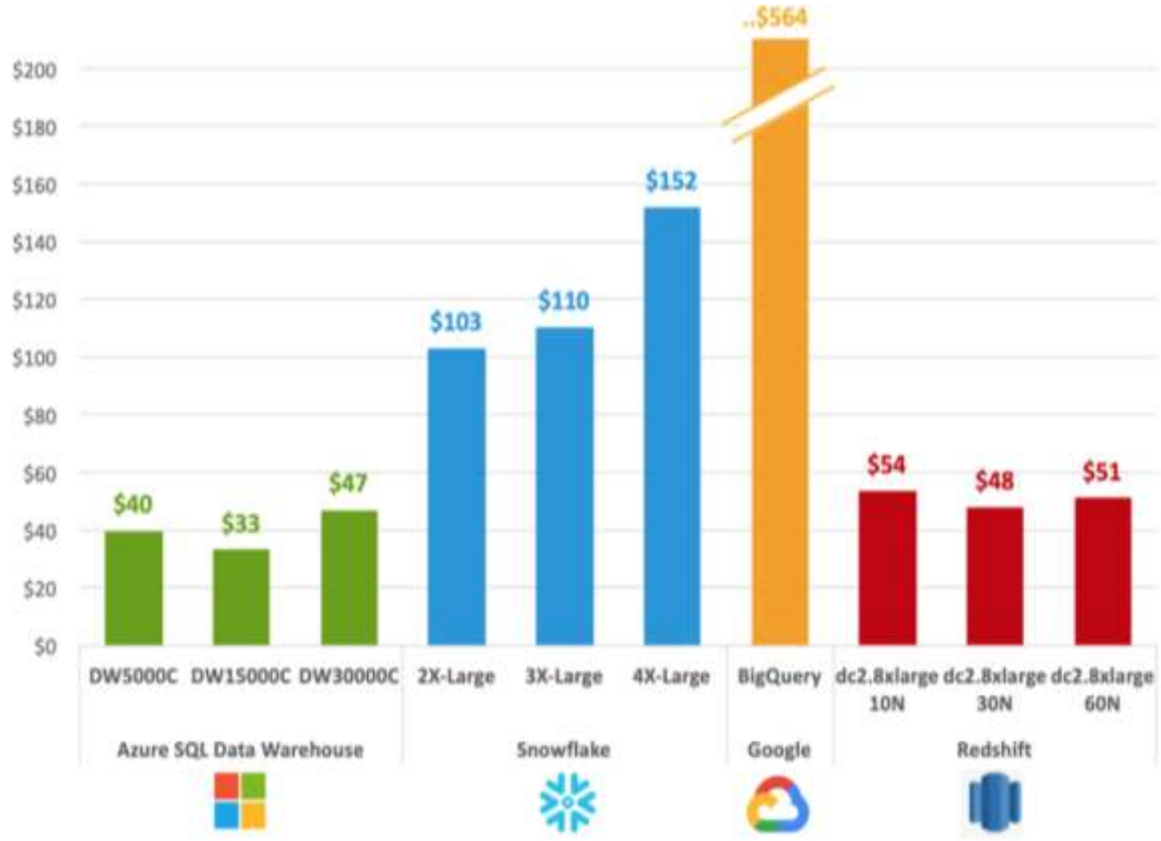


Microsoft

SQL DW Gen2: New Features & Enhancements

Kal Yella

Best-in-class price per performance



Price-performance is calculated by GigaOm as the TPC-H metric of cost of ownership divided by composite query.

Results based on GigaOm's TPC-H results, published in January 2019

Agenda

Adaptive cache internals

Instant data movement on Gen2

Resource classes on Gen2

Workload Management

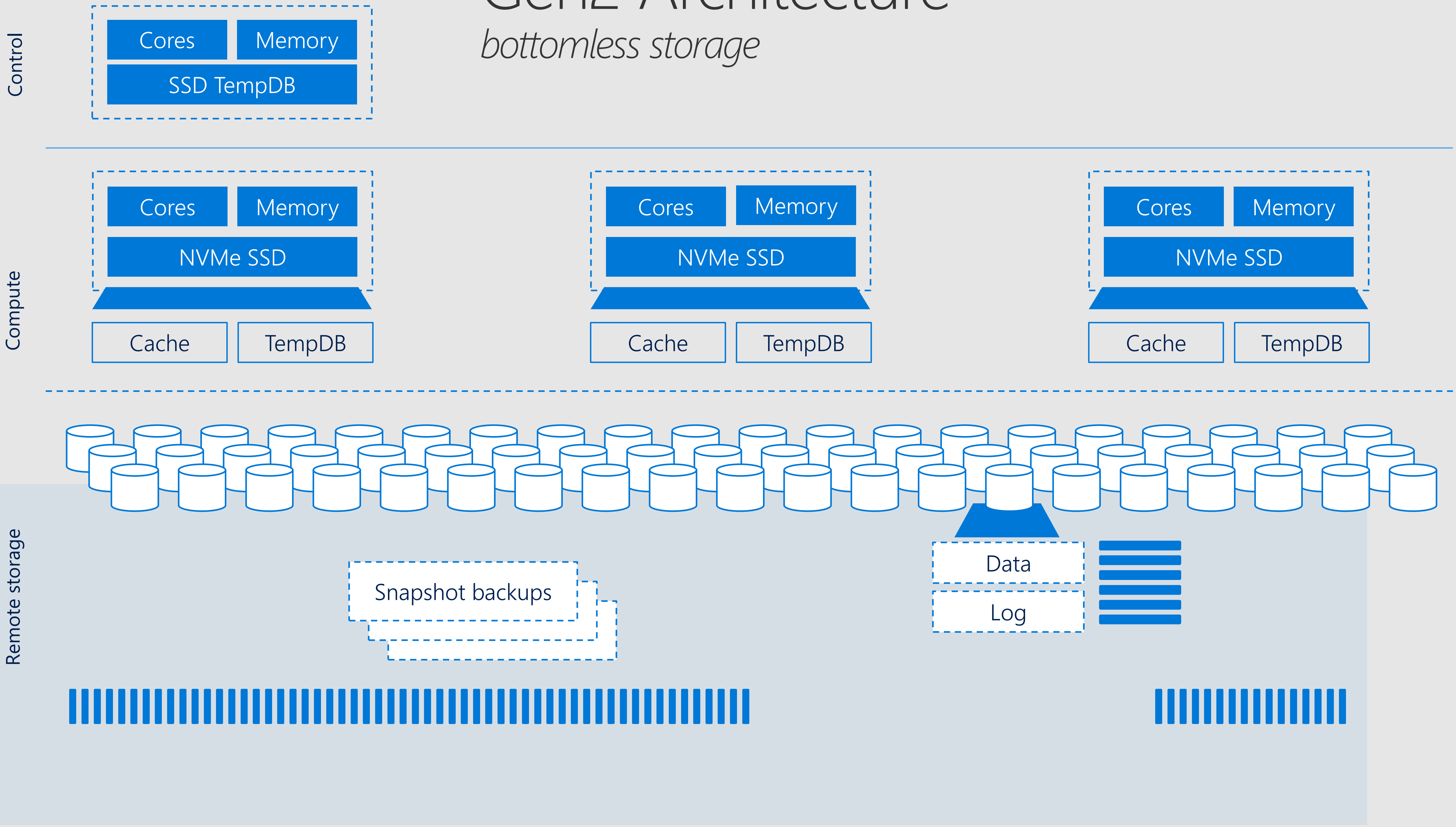
Security Enhancements

What’s New since Gen2

Adaptive cache

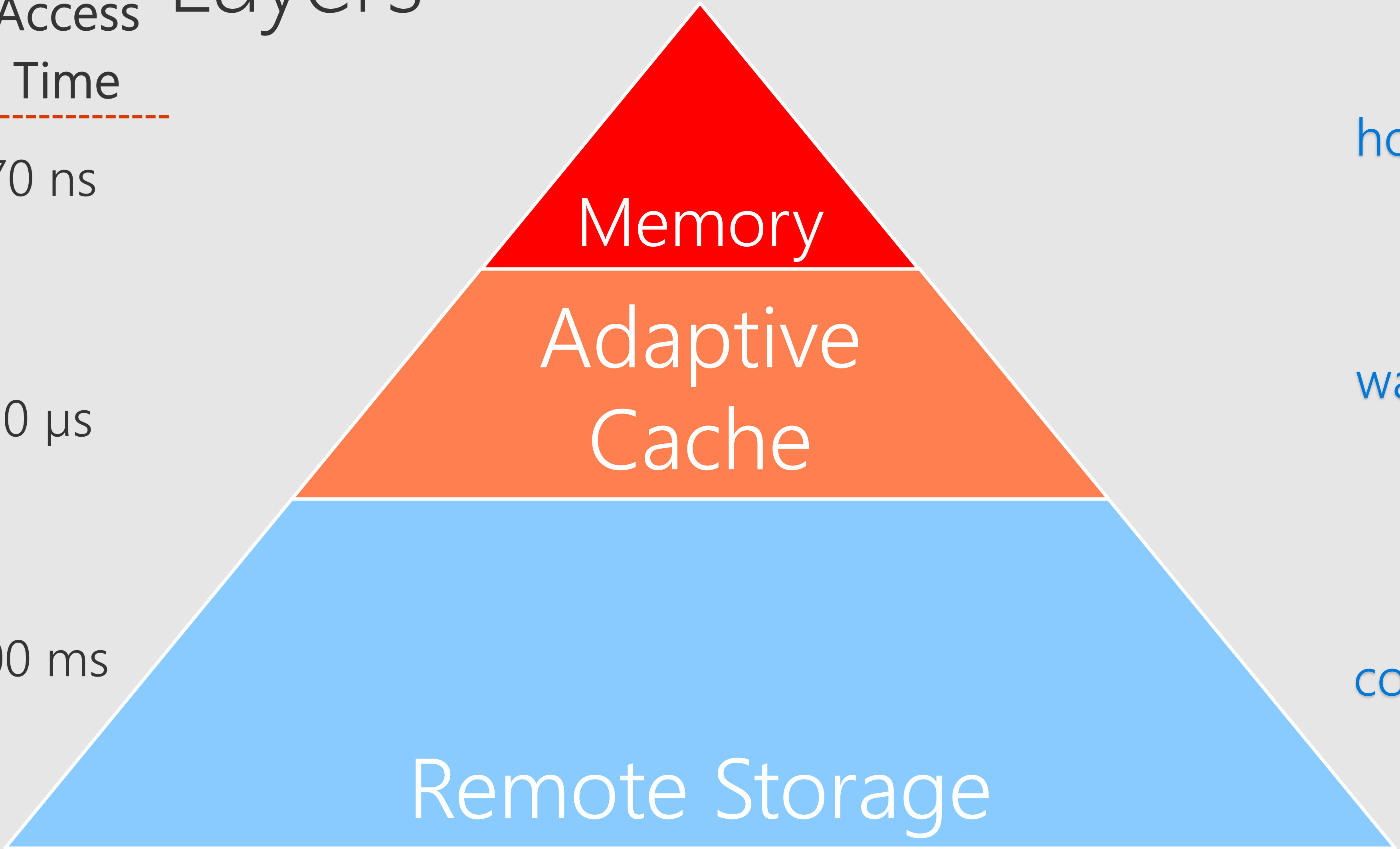
Gen2 Architecture

bottomless storage



Automated Tiering Of Storage Layers

Node Capacity	Access Time
300GB+	70 ns
1.5TB	20 µs
∞	100 ms



Memory

Adaptive Cache

Remote Storage

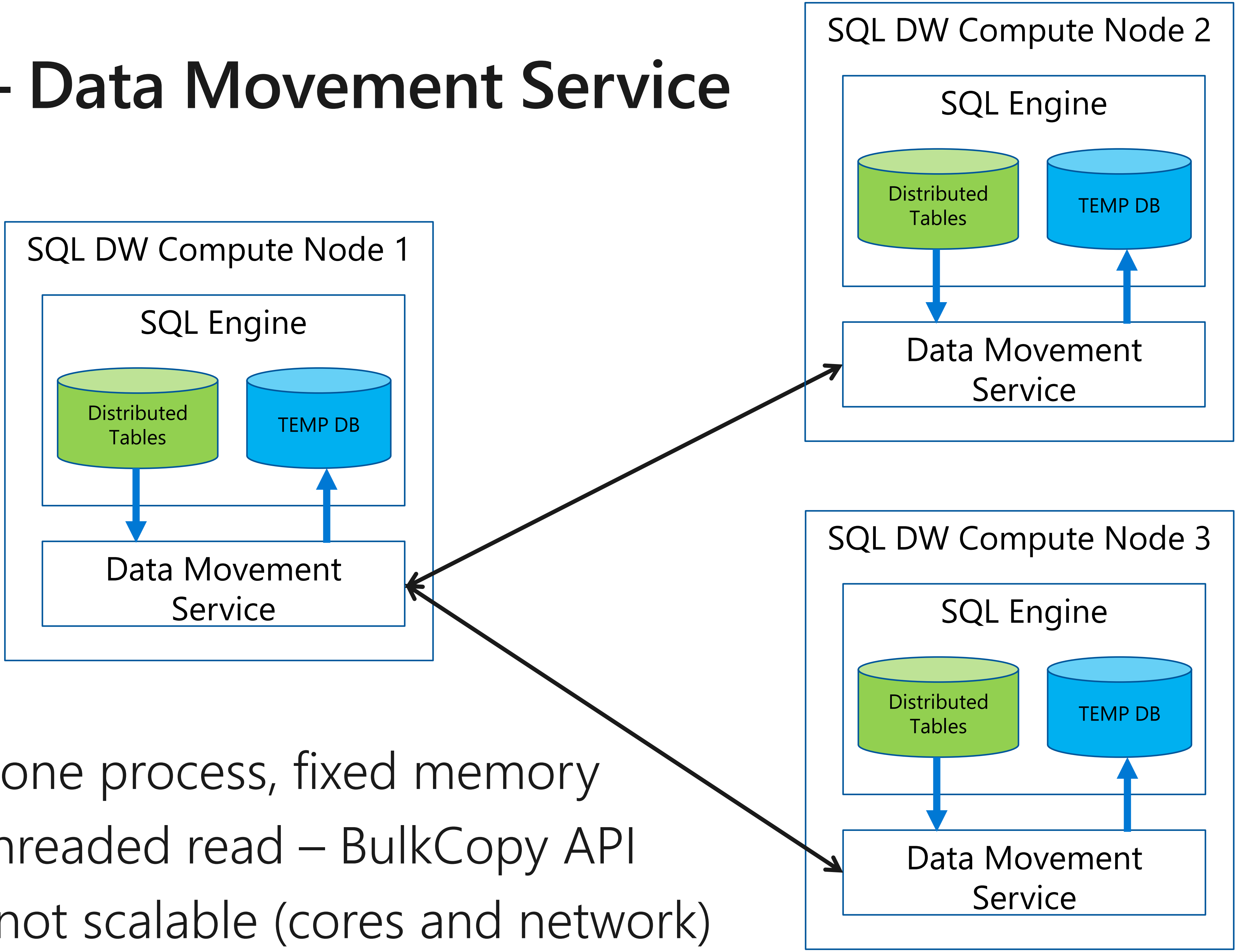
hot data

warm data

cold data

Gen2 Performance Accelerator

Gen1 – Data Movement Service

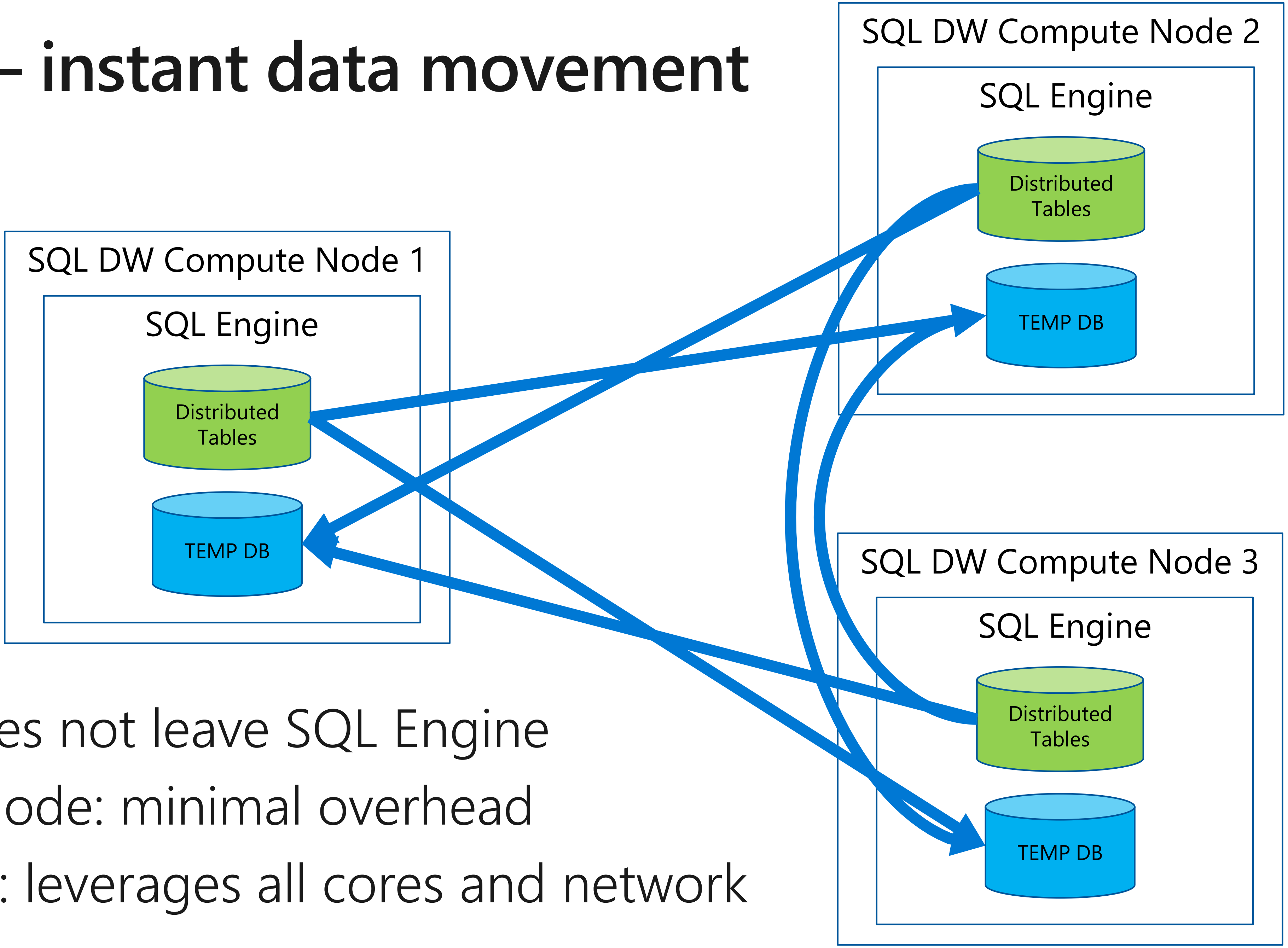


Stand-alone process, fixed memory

Single-threaded read – BulkCopy API

Copy is not scalable (cores and network)

Gen2 – instant data movement

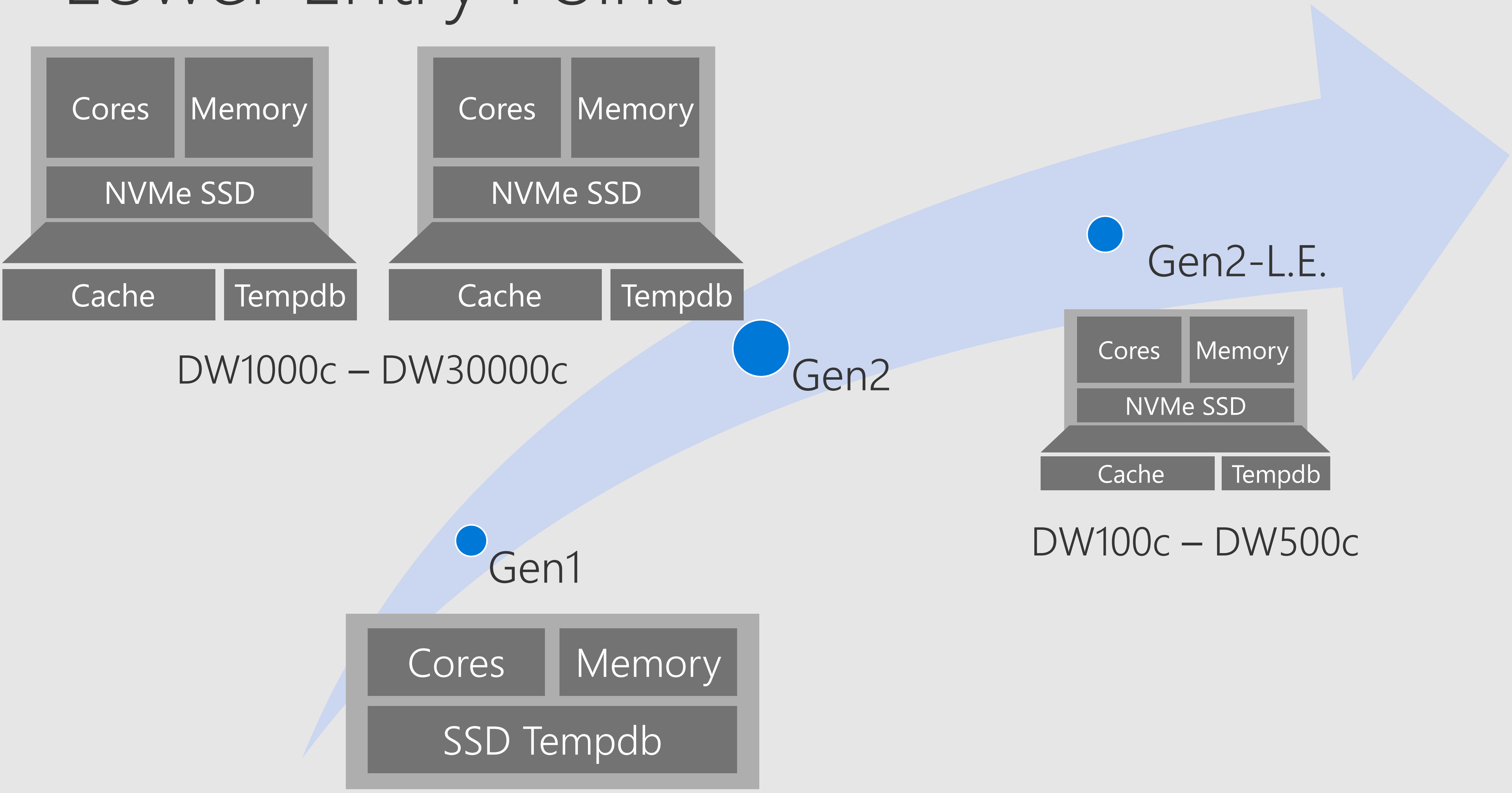


Data does not leave SQL Engine

Batch-mode: minimal overhead

Scalable: leverages all cores and network

Lower Entry Point



DW1000c – DW30000c

DW100c – DW500c

DW100 - DW6000

Gen2 Resource Classes

Gen2 – Simpler resource model

With Gen2, dynamic resource pools were introduced with a 3-10-22-70 model for resource allocations.

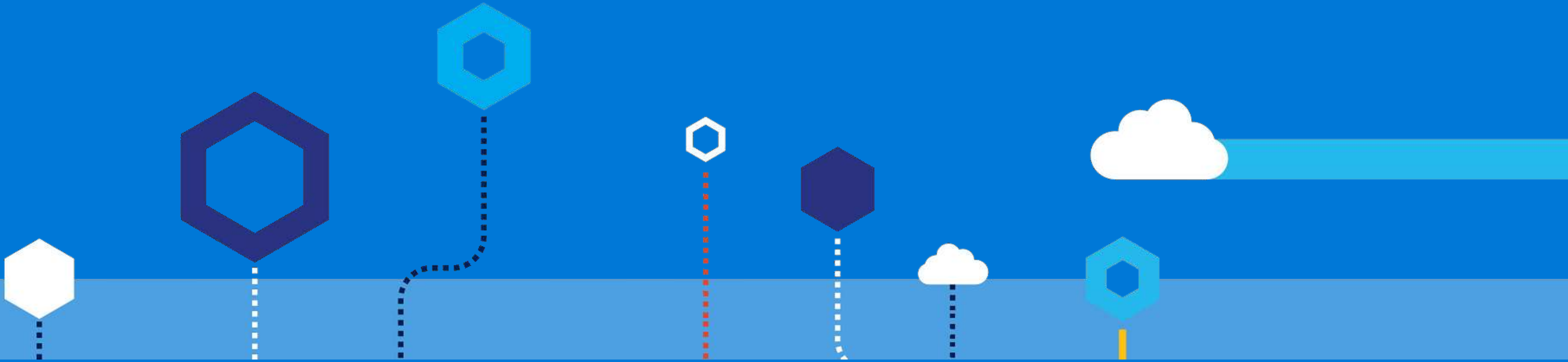
Resource Class	Percent Resources	Concurrency
SmallRc	3%	32
MediumRc	10%	10
LargeRc	22%	4
XLargeRc	70%	1

Gen2 – Simpler model

At MediumRc, an example…

MediumRC		Gen1 Model		Gen2 Model	
Service Level	Total Slots	Slots	Concurrency	Slots	Concurrency
DW1000c	40	8	5	4	10
DW1500c	60	8	7	6	10
DW2000c	80	16	5	8	10
DW2500c	100	16	6	10	10
DW3000c	120	16	7	12	10
DW5000c	200	32	6	20	10
DW6000c	240	32	7	24	10
DW7500c	300	64	4	30	10
DW10000c	400	64	6	40	10
DW15000c	600	64	9	60	10
DW30000c	1200	64	18	120	10

What's new since Gen2



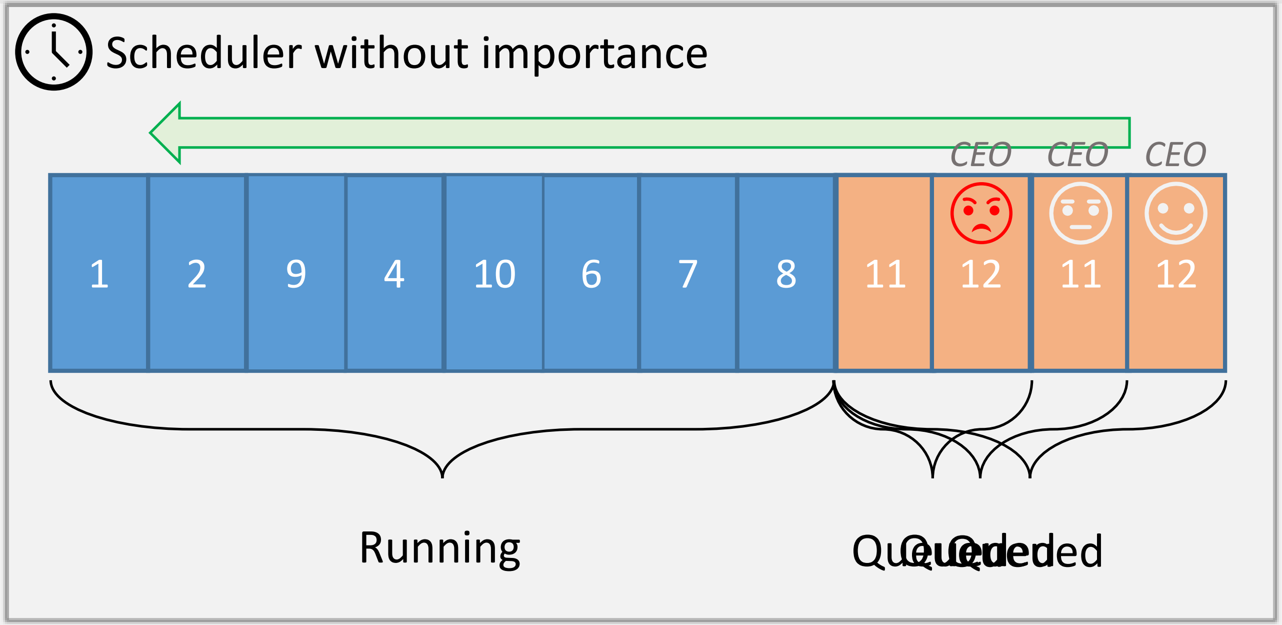
Workload Management

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

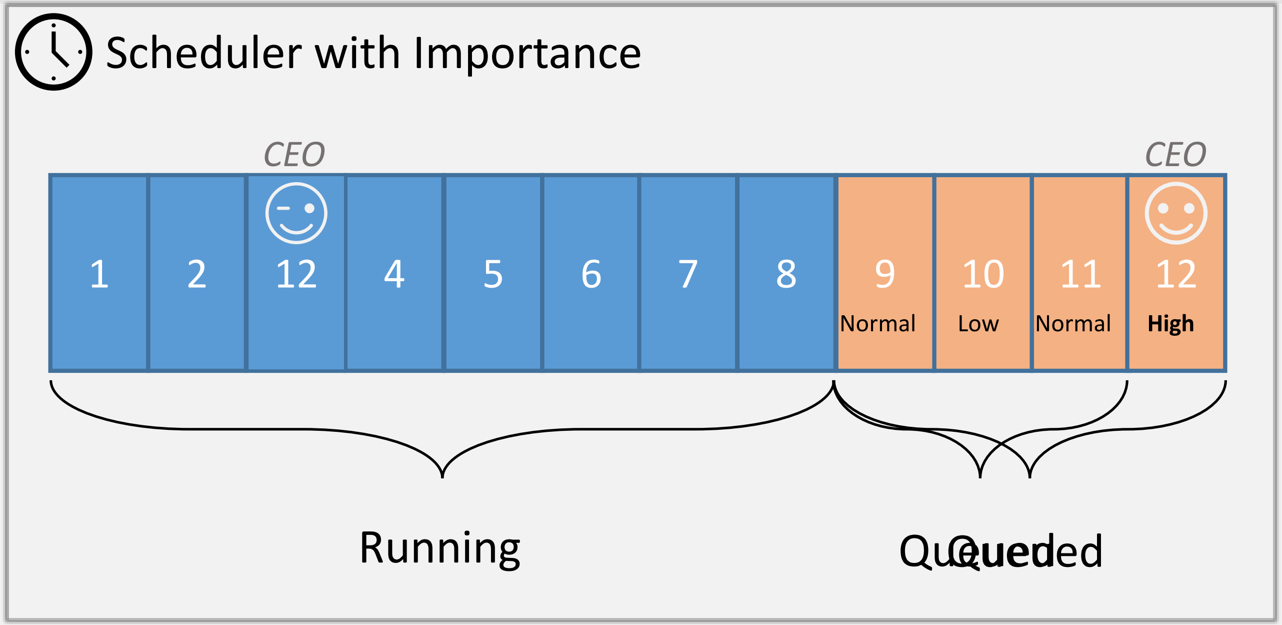
```
CREATE WORKLOAD CLASSIFIER classifier_name
WITH
(
   [WORKLOAD_GROUP = '<Resource Class>' ]
   [IMPORTANCE = {    LOW            |
                      BELOW_NORMAL   |
                      NORMAL         |
                      ABOVE_NORMAL   |
                      HIGH
                 }
   ]
   [MEMBERNAME = ‘security_account’]
)
```

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.



Scheduler with Importance

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 ]
    [ MAX_CONCURRENCY = value ]
)
```

Gen2 Performance Announcement Details

Gen2 GA

[Turbocharge cloud analytics with Azure SQL Data Warehouse]

[Blazing fast data warehousing with Azure SQL Data Warehouse]

Adaptive Caching

[Adaptive caching powers Azure SQL Data Warehouse performance gains]

Instant data movement

[Lightning fast query performance with Azure SQL Data Warehouse]

GigaOm Benchmarking Report

https://gigaom.com/report/data-warehouse-in-the-cloud-benchmark/

Security Enhancements

Row-level security (RLS)

Overview

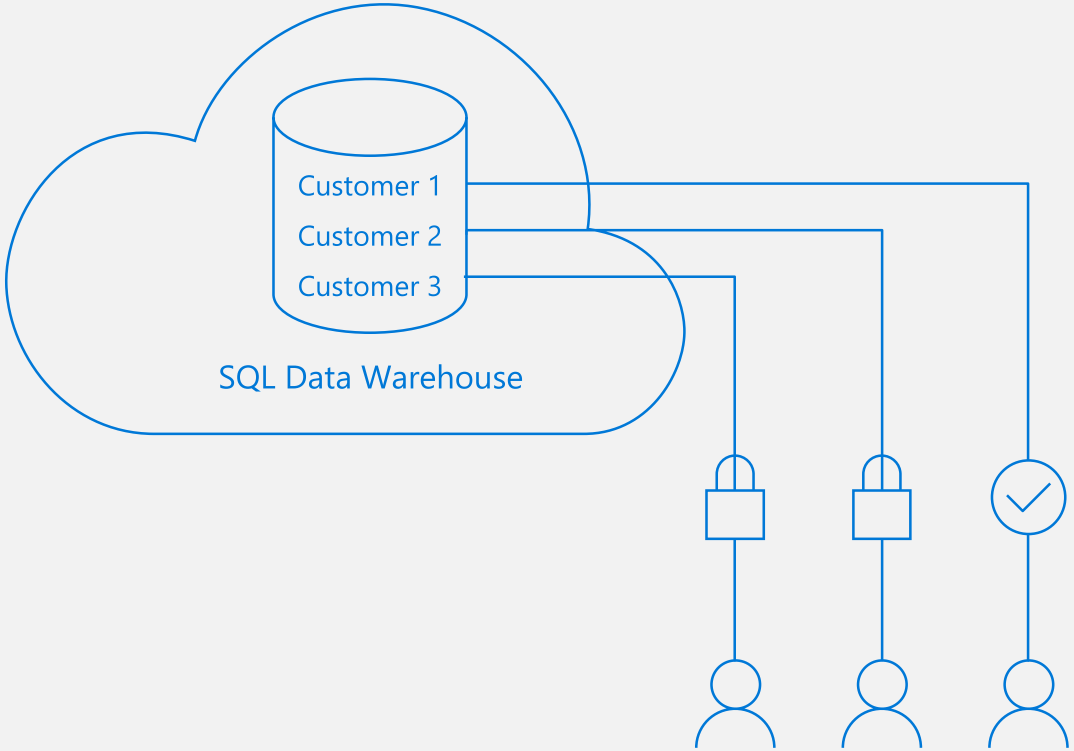
Fine grained access control of specific rows in a database table

Help prevent unauthorized access when multiple users share the same tables

Eliminates need to implement connection filtering in multitenant applications

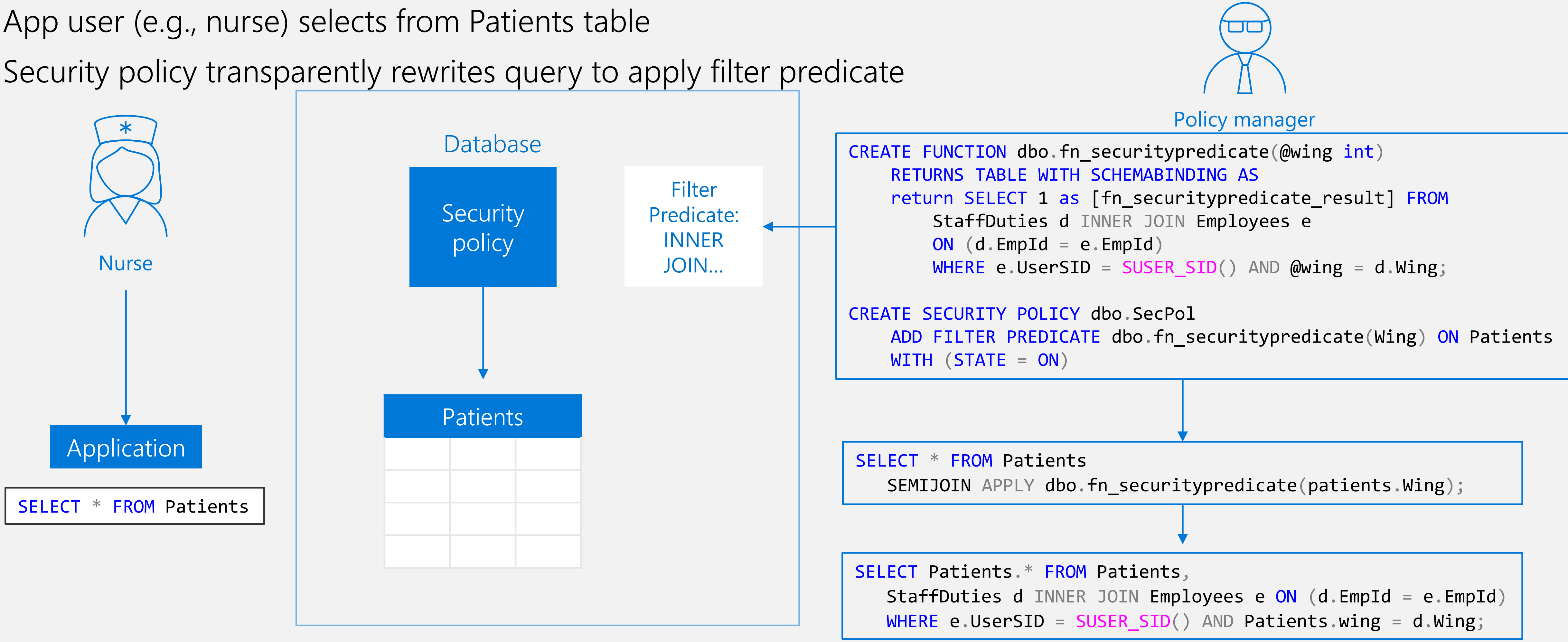
Administer via SQL Server Management Studio or SQL Server Data Tools

Easily locate enforcement logic inside the database and schema bound to the table



RLS in three steps

1. Policy manager creates filter predicate and security policy in T-SQL, binding the predicate to the Patients table
2. App user (e.g., nurse) selects from Patients table
3. Security policy transparently rewrites query to apply filter predicate



```
SELECT * FROM Patients
```

```
CREATE FUNCTION dbo.fn_securitypredicate(@wing int)
    RETURNS TABLE WITH SCHEMABINDING AS
    return SELECT 1 as [fn_securitypredicate_result] FROM
        StaffDuties d INNER JOIN Employees e
        ON (d.EmpId = e.EmpId)
        WHERE e.UserSID = SUSER_SID() AND @wing = d.Wing;

CREATE SECURITY POLICY dbo.SecPol
    ADD FILTER PREDICATE dbo.fn_securitypredicate(Wing) ON Patients
    WITH (STATE = ON)
```

```
SELECT * FROM Patients
   SEMIJOIN APPLY dbo.fn_securitypredicate(patients.Wing);
```

```
SELECT Patients.* FROM Patients,
   StaffDuties d INNER JOIN Employees e ON (d.EmpId = e.EmpId)
   WHERE e.UserSID = SUSER_SID() AND Patients.wing = d.Wing;
```

COLUMN-level security (CLS)

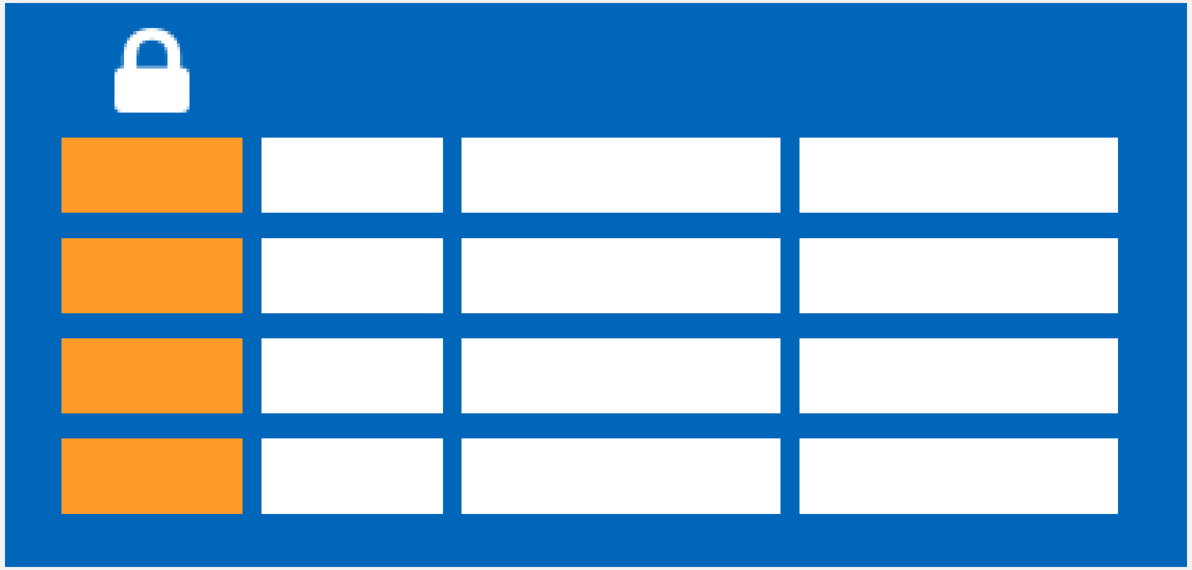
Overview

Control access of specific columns in a database table based on customer's group membership or execution context

Simplifies the design and implementation of security by putting restriction logic in database tier as opposed to application tier

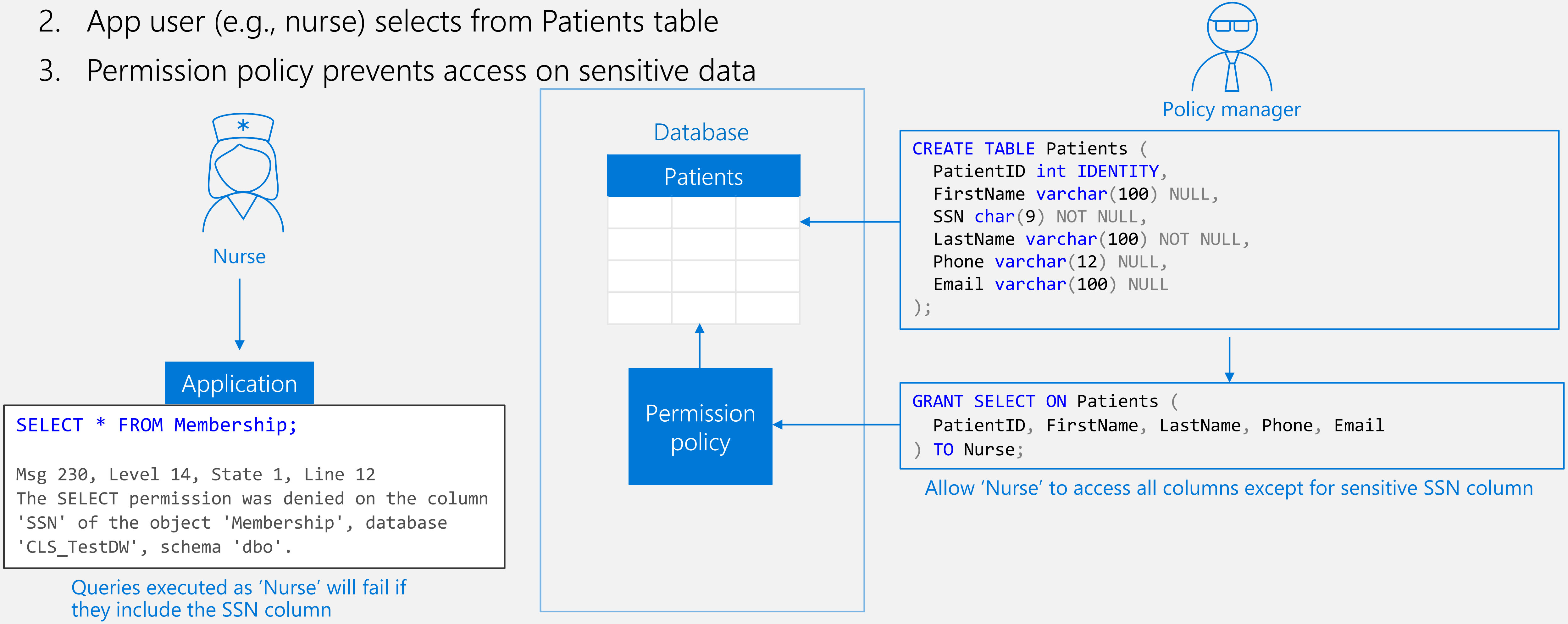
Administer via GRANT T-SQL statement

Both Azure Active Directory (AAD) and SQL authentication are supported



CLS in three steps

1. Policy manager creates permission policy in T-SQL, binding the policy to the Patients table on a specific group
2. App user (e.g., nurse) selects from Patients table
3. Permission policy prevents access on sensitive data



```sql
CREATE TABLE Patients (
  PatientID int IDENTITY,
  FirstName varchar(100) NULL,
  SSN char(9) NOT NULL,
  LastName varchar(100) NOT NULL,
  Phone varchar(12) NULL,
  Email varchar(100) NULL
);
```

```sql
GRANT SELECT ON Patients (
  PatientID, FirstName, LastName, Phone, Email
) TO Nurse;
```

Allow ‘Nurse’ to access all columns except for sensitive SSN column

```
SELECT * FROM Membership;

Msg 230, Level 14, State 1, Line 12
The SELECT permission was denied on the column
'SSN' of the object 'Membership', database
'CLS_TestDW', schema 'dbo'.
```

Queries executed as ‘Nurse’ will fail if they include the SSN column

SQL Data discovery & classification PRE

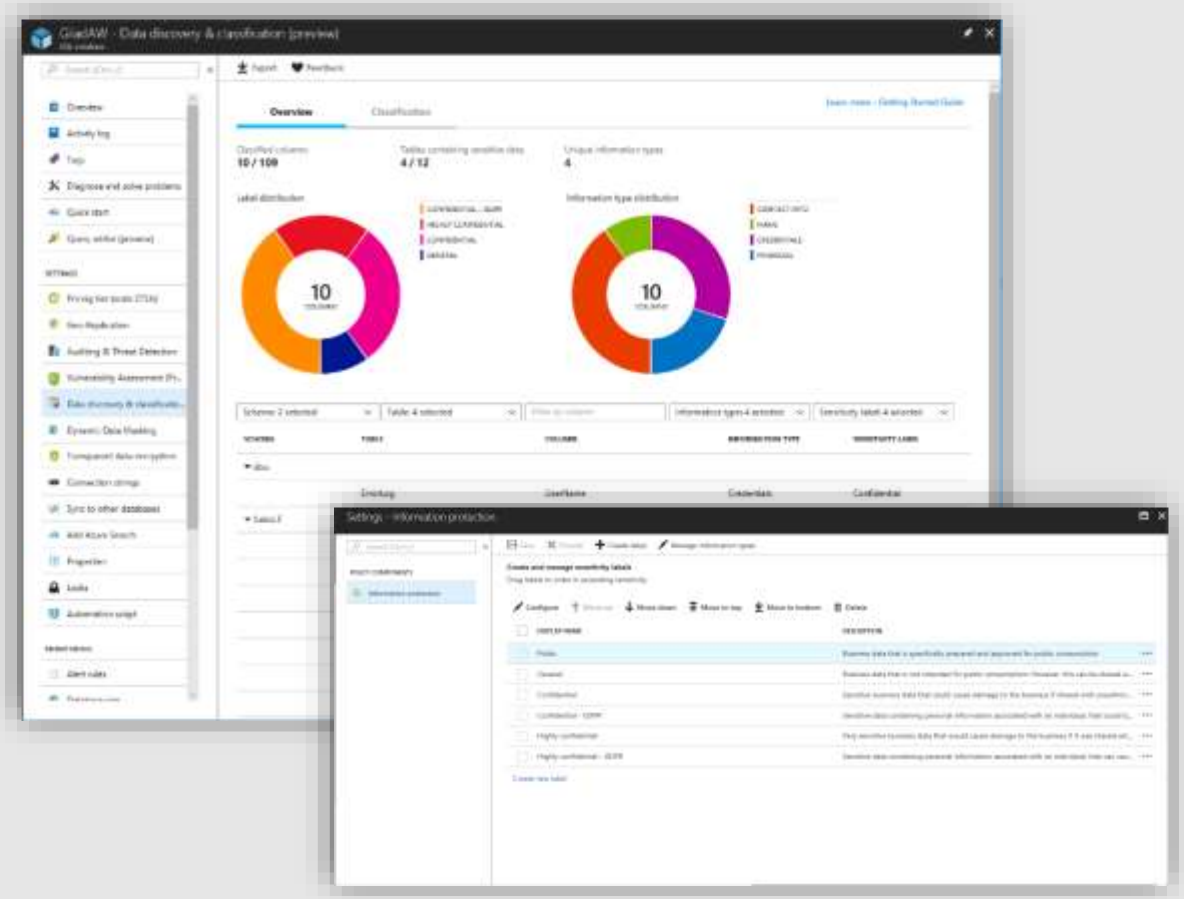
Overview

Automatic discovery of columns with sensitive data

Add persistent sensitive data labels

Audit and detect access to the sensitive data

Manage labels for your entire Azure tenant using Azure Security Center


Data discovery & classification (preview)
Overview
Classification
10 / 109
4 / 12
4
10
10
Settings - Information protection

Discover

Classify

Label

DATA WAREHOUSE MANAGED IDENTITIES

Overview

Azure SQL Data Warehouse supports managed service identity authentication to Azure Data Lake. This removes the need for storing access credentials in code or in Azure Key vault.

```
 # Generate and assign an Azure AD Identity for DW
Set-AzureRmSqlServer -ResourceGroupName
$resourceGroupName -ServerName $serverName -
AssignIdentity

# Get ServicePrincipalId assigned to DW
$serverAzureAdIdentity =
(Get-AzureRmADServicePrincipal -SearchString
$serverName).Id
```

Result-set caching

Overview

Cache the results of a query in DW storage. This enables interactive response times for repetitive queries against tables with infrequent data changes.

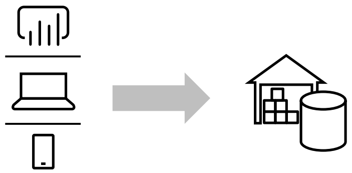
The result-set cache persists even if a data warehouse is paused and resumed later.

Query cache is invalidated and refreshed when underlying table data or query code changes.

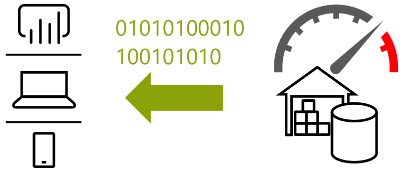
Result cache is evicted regularly based on a time-aware least recently used algorithm (TLRU).

```sql
-- 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
-- Run on target data warehouse
SELECT *
FROM   sys.dm_pdw_request_steps
WHERE  command like '%DWResultCacheDb%'
       AND step_index = 0
```

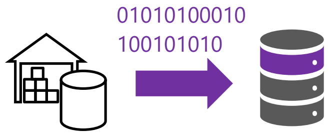
Result-set caching flow



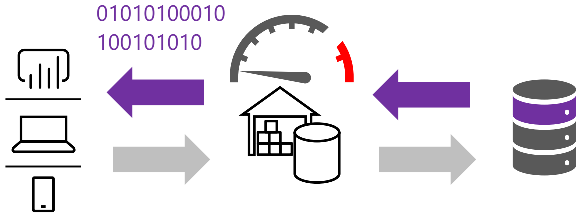
1 Client sends query to DW



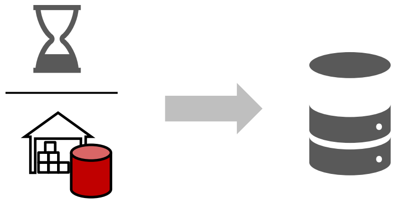
2 Query is processed using DW compute nodes which pull data from remote storage, process query and output back to client app



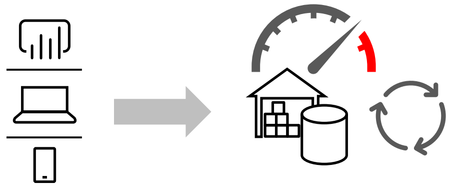
3 Query results are cached in remote storage so subsequent requests can be served immediately



4 Subsequent executions for the same query bypass compute nodes and can be fetched instantly from persistent cache in remote storage



5 Remote storage cache is evicted regularly based on time, cache usage, and any modifications to underlying table data.



6 Cache will need to be regenerated if query results have been evicted from cache

Indexed (materialized) views

Overview

Indexed views cache the schema and data for a view in DW remote storage. They are useful for improving the performance of 'SELECT' statement queries that include aggregations

Indexed views are automatically updated when data in underlying tables are changed. This is a synchronous operation that occurs as soon as the data is 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

```
-- 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;
```

Indexed (materialized) views - example

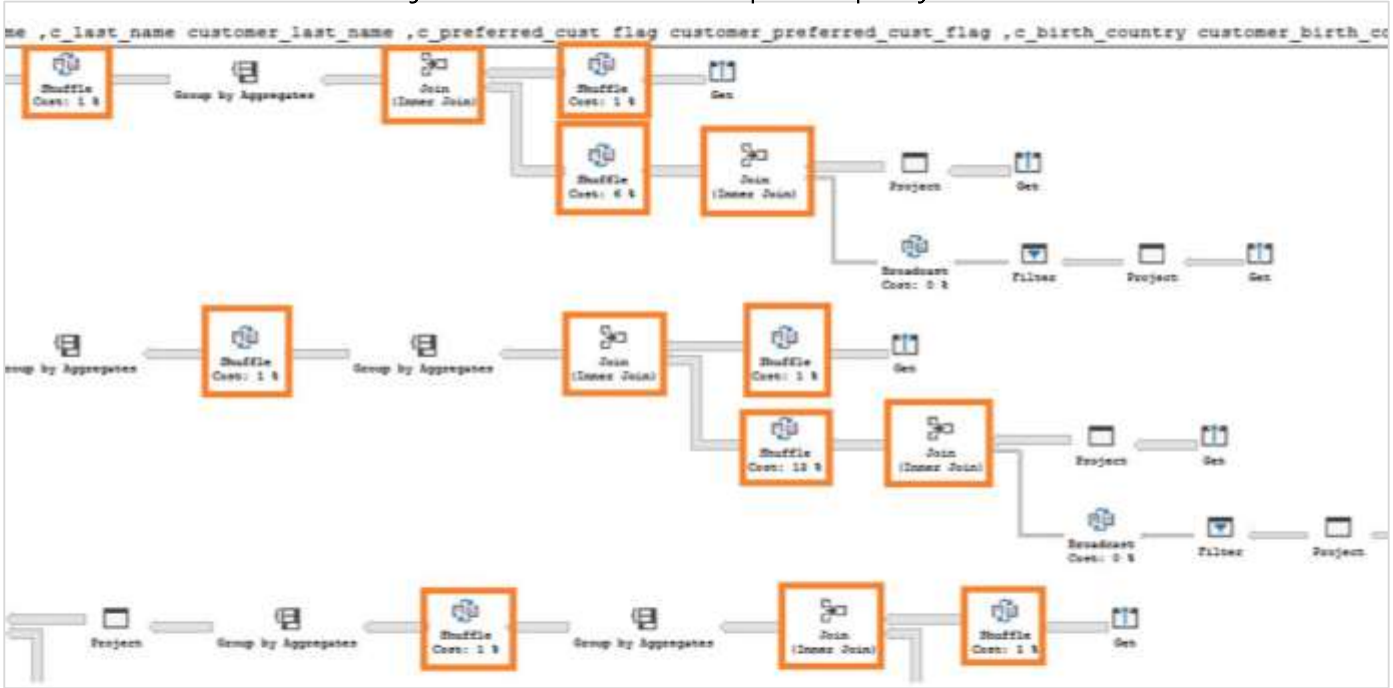
In this example, a query to get the year total sales per customer is shown to have a lot of data shuffles and joins that contribute to slow performance:

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



me ,c_last_name customer_last_name ,c_preferred_cust_flag customer_preferred_cust_flag ,c_birth_country customer_birth_co

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

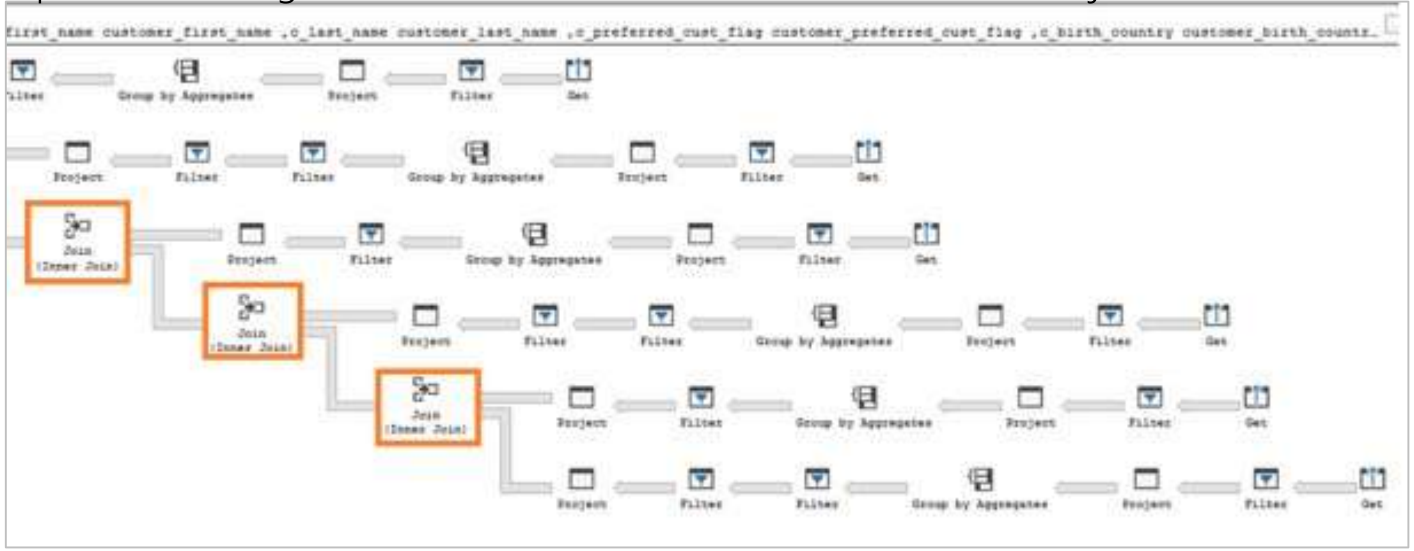
The SQL Data Warehouse query optimizer automatically leverages the indexed view to speed up the same query. Notice that the query does not need to reference the view directly

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



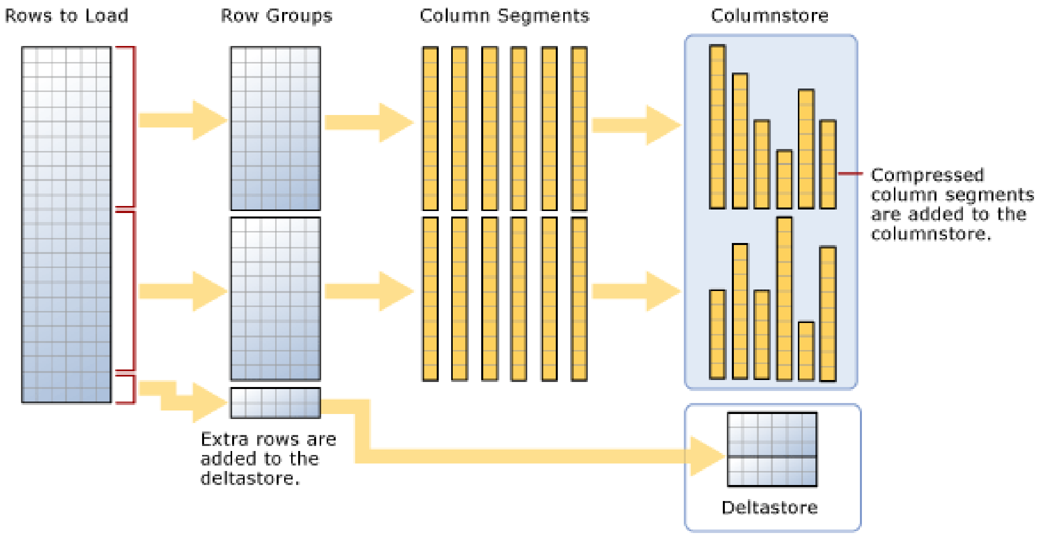
```
first_name customer_first_name ,c_last_name customer_last_name ,c_preferred_cust_flag customer_preferred_cust_flag ,c_birth_country customer_birth_countr…
```

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.


Rows to Load
Row Groups
Column Segments
Columnstore
Compressed column segments are added to the columnstore.
Extra rows are added to the deltastore.
Deltastore

```
-- 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
```

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

```sql
-- 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 }}]'

```sql
-- 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

ACCELERATED DATABASE RECOVERY

Overview

New SQL database engine feature

Redesigns the engine recovery process

Speeds up the 3 recovery phases

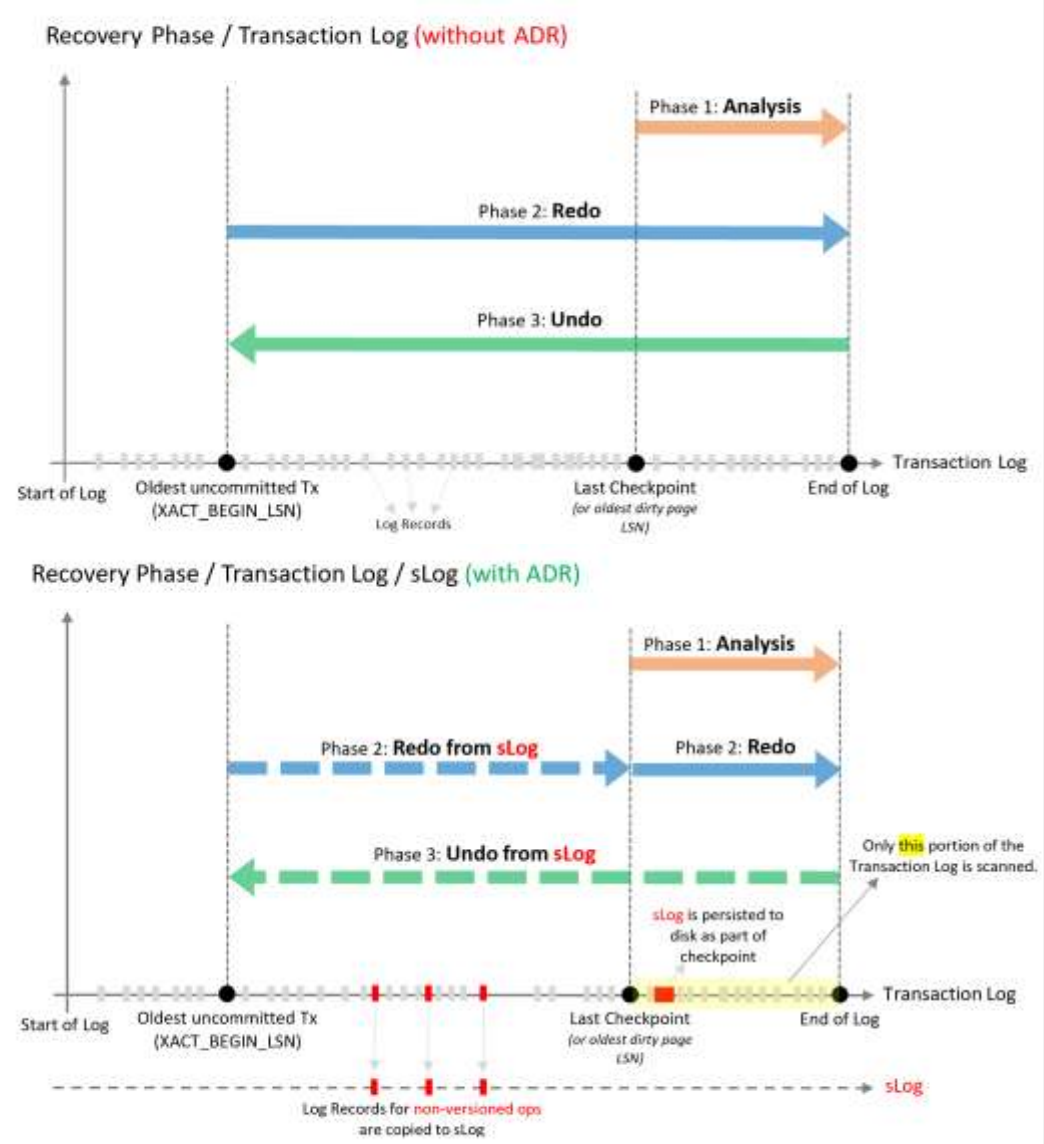
Benefits

Fast and consistent database recovery

Instantaneous transaction rollback

Regardless of number or age of transactions

Fast pause and resume operations


Recovery Phase / Transaction Log (without ADR)
Phase 1: Analysis
Phase 2: Redo
Phase 3: Undo
Transaction Log
Start of Log
Oldest uncommitted Tx (XACT_BEGIN_LSN)
Log Records
Last Checkpoint (or oldest dirty page LSN)
End of Log
Recovery Phase / Transaction Log / sLog (with ADR)
Phase 1: Analysis
Phase 2: Redo from sLog
Phase 2: Redo
Phase 3: Undo from sLog
Only this portion of the Transaction Log is scanned.
sLog is persisted to disk as part of checkpoint
Transaction Log
Start of Log
Oldest uncommitted Tx (XACT_BEGIN_LSN)
Last Checkpoint (or oldest dirty page LSN)
End of Log
sLog
Log Records for non-versioned ops are copied to sLog

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

```sql
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

Approximate execution

HyperLogLog accuracy

Will return a result with a 2% accuracy of true cardinality on average.

e.g. COUNT (DISTINCT) returns 1,000,000, HyperLogLog will return a value in the range of 999,736 to 1,016,234.

APPROX_COUNT_DISTINCT

Returns the approximate number of unique non-null values in a group.

Use Case: Approximating web usage trend behavior

```sql
-- Syntax
APPROX_COUNT_DISTINCT ( expression )

-- The approximate number of different order keys by order status from the orders table.
SELECT O_OrderStatus, APPROX_COUNT_DISTINCT(O_OrderKey) AS Approx_Distinct_OrderKey
FROM dbo.Orders
GROUP BY O_OrderStatus
ORDER BY O_OrderStatus;
```

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.

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

```sql
-- 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()
```

Since Gen2 GA

05/14/18 Automatic Statistics
06/16/18 User Defined Restore Points
06/21/19 Column-level security
07/25/18 Fast Restore
08/02/18 Recommendations for data skew and table statistics
09/24/18 Streaming support in Azure Databricks
09/24/18 User defined maintenance scheduling
09/24/18 Vulnerability assessment
09/24/18 Intelligent Insights
09/24/18 Flexible Restore Points
11/07/18 RLS
11/12/18 Azure Monitor log support
12/08/18 Azure Virtual Network service endpoints
01/10/19 Azure Data Box Disk

Upcoming Features:
Copy Command
Work Load Management

Microsoft

Q&A

Microsoft

Modernizing Your Data Warehouse