
SQL
SQL Data Warehouse
Azure SQL DataWarehouse Performance and Maintenance
Presented by Robin Lester
Microsoft

Where does SQL Data Warehouse fit?

SQL Server VM (Iaas)	Azure SQL Database	Azure SQL Data Warehouse	Azure Data Lake
OLTP / DW workloads Lift and Shift Customer managed 1-1TB+	OLTP/ DW workloads Net new development Fully managed service 1-4000 GB	DW workloads only Fully managed Dynamic Pause/Scale 250GB – PB+	Non-relational Cheap, flexible Access Processing raw data 1 TB+

The move to the cloud

Parallel Data Warehouse

£250,000 -> £1 Mill

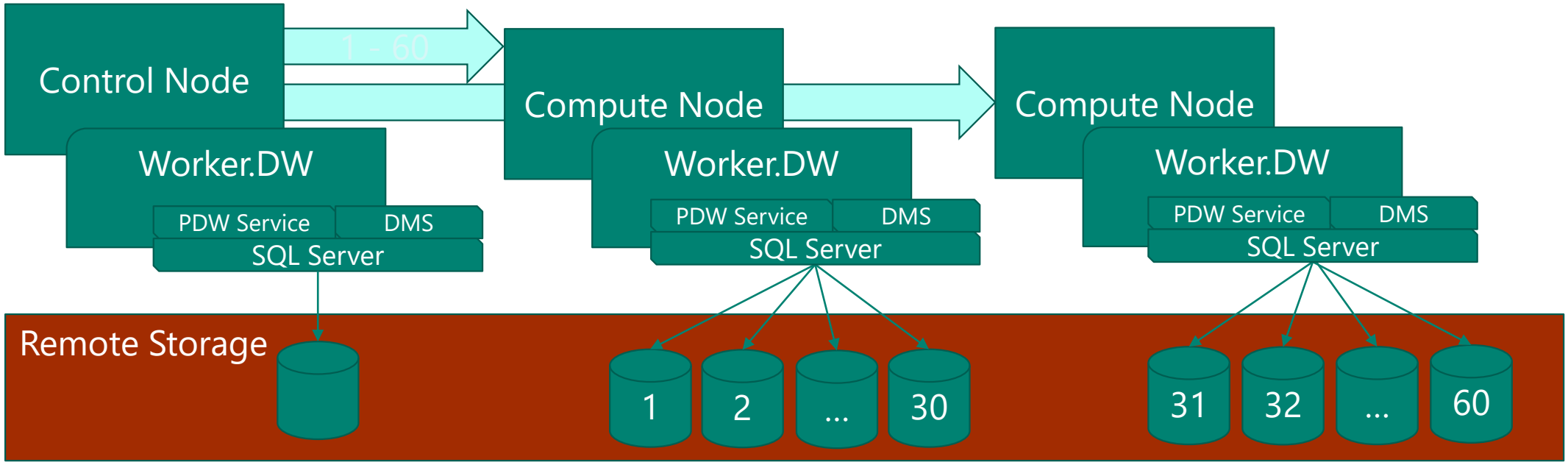
Analytics Platform System

Azure SQL Data Warehouse

From £320 PCM

Architecture

- One control node
- 1-60 compute nodes
- 60 Databases



Service Level Objectives

- ASDW has 60 SQL Azure Databases. The higher the scale the larger number of separate co
pl
- Da
detach and attach when scaling

DWU	Scan Rate	Load Rate	Cost
100	1M Rows/sec	15K Rows/Sec	1x
200	2M Rows/sec	30K Rows/Sec	2x
300	3M Rows/sec	45K Rows/sec	3x

	DTU	# of Compute nodes	# of distributions
DW100	750	1	60
DW200	1500	2	60
			0
			0
			0
			0
DW1000	7500	10	60
DW1200	9000	12	60
DW1500	11250	15	60
DW2000	15000	20	60
DW3000	22500	30	60
DW6000	45000	60	60

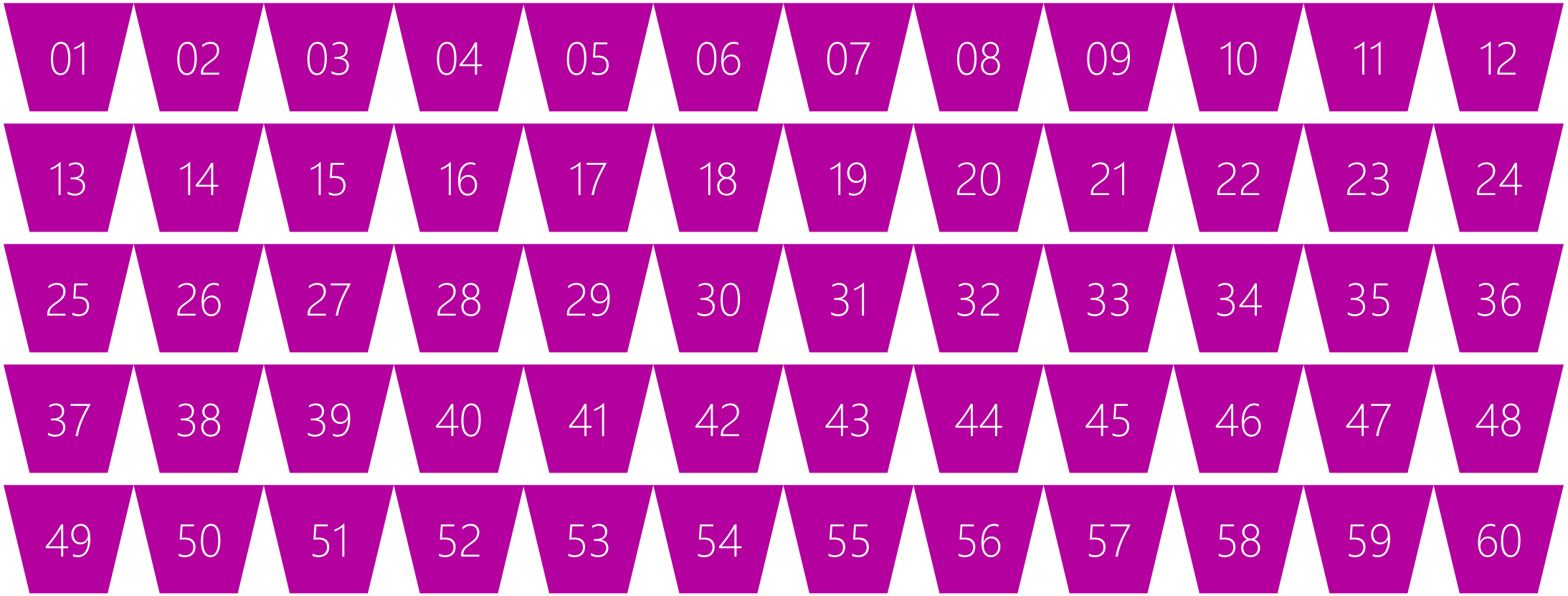
To calculate the number of compute nodes divide DWU by 100

6000/100 = 60

1200/100 = 12

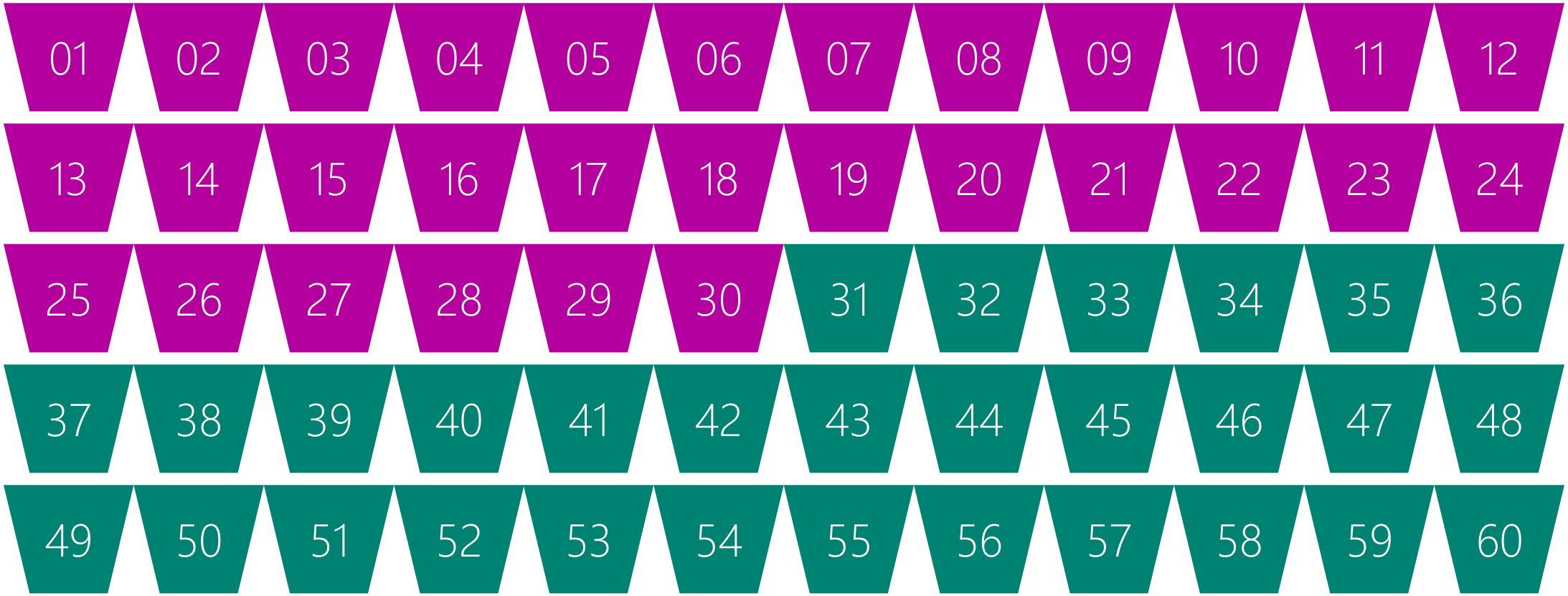
Mapping Compute in SQLDW

DW100



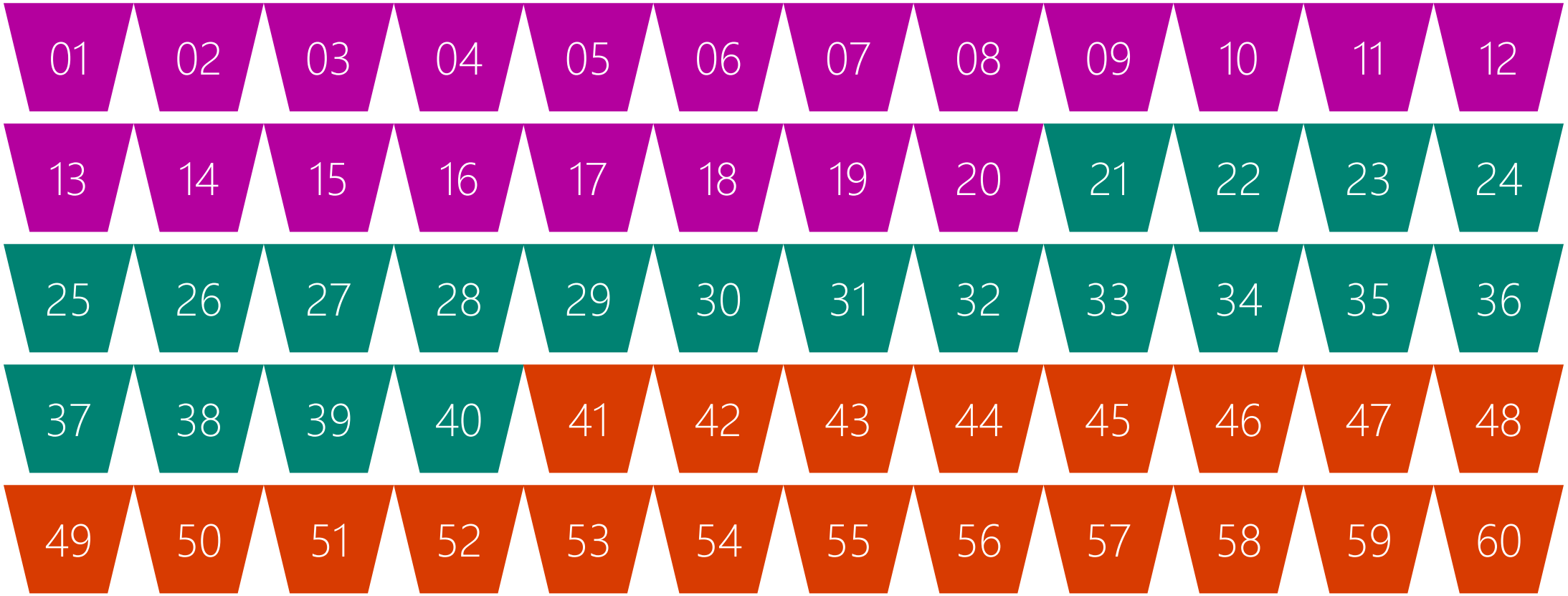
Mapping Compute in SQLDW

DW200



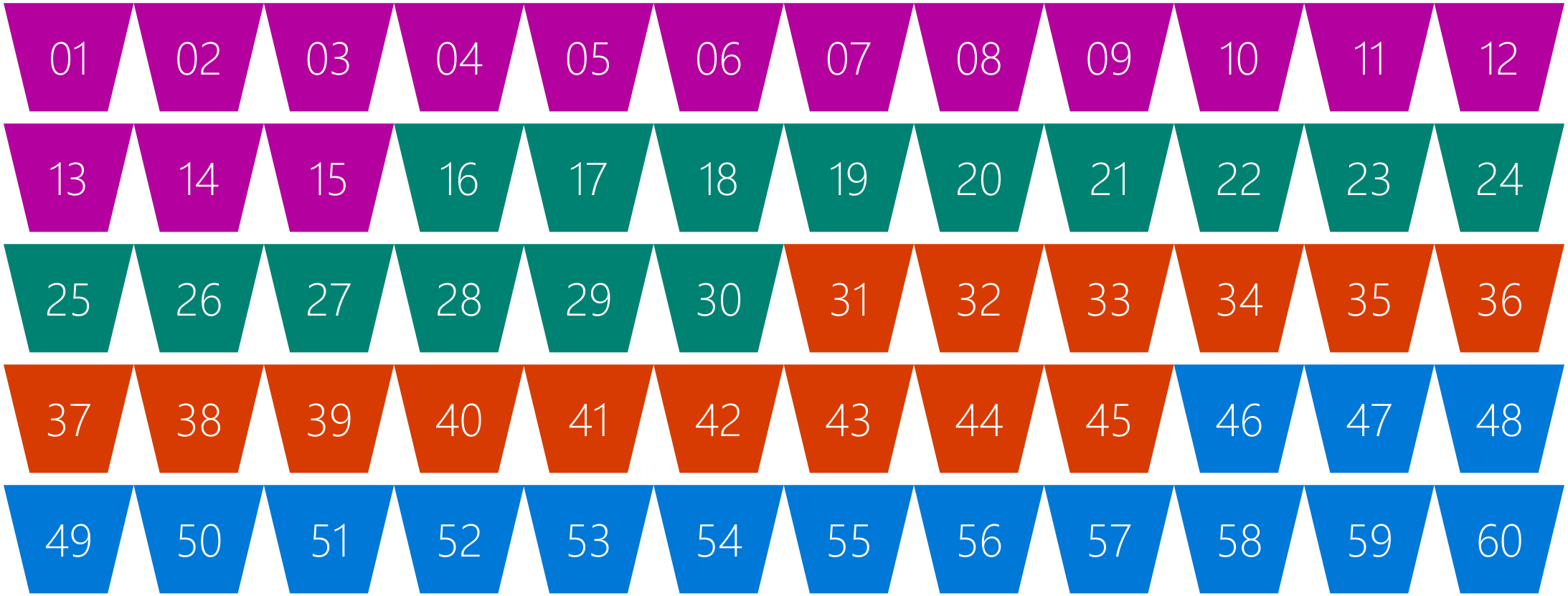
Mapping Compute in SQLDW

DW300



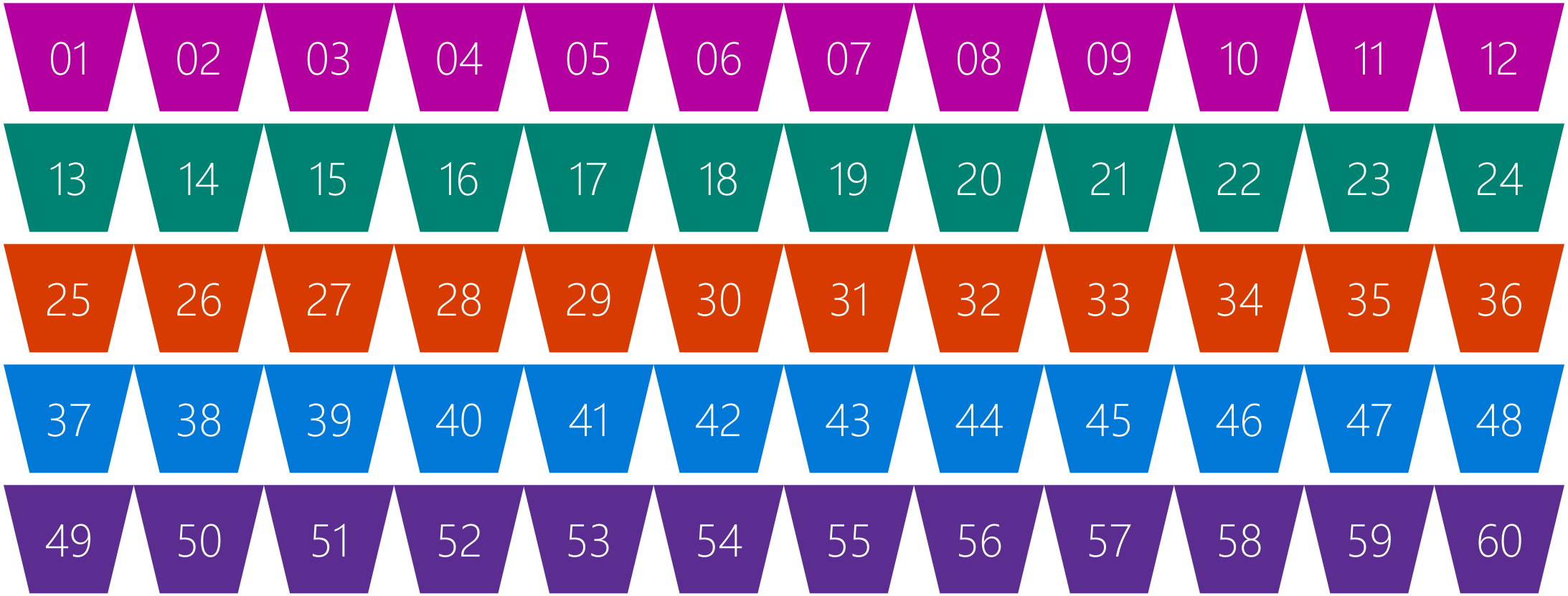
Mapping Compute in SQLDW

DW400

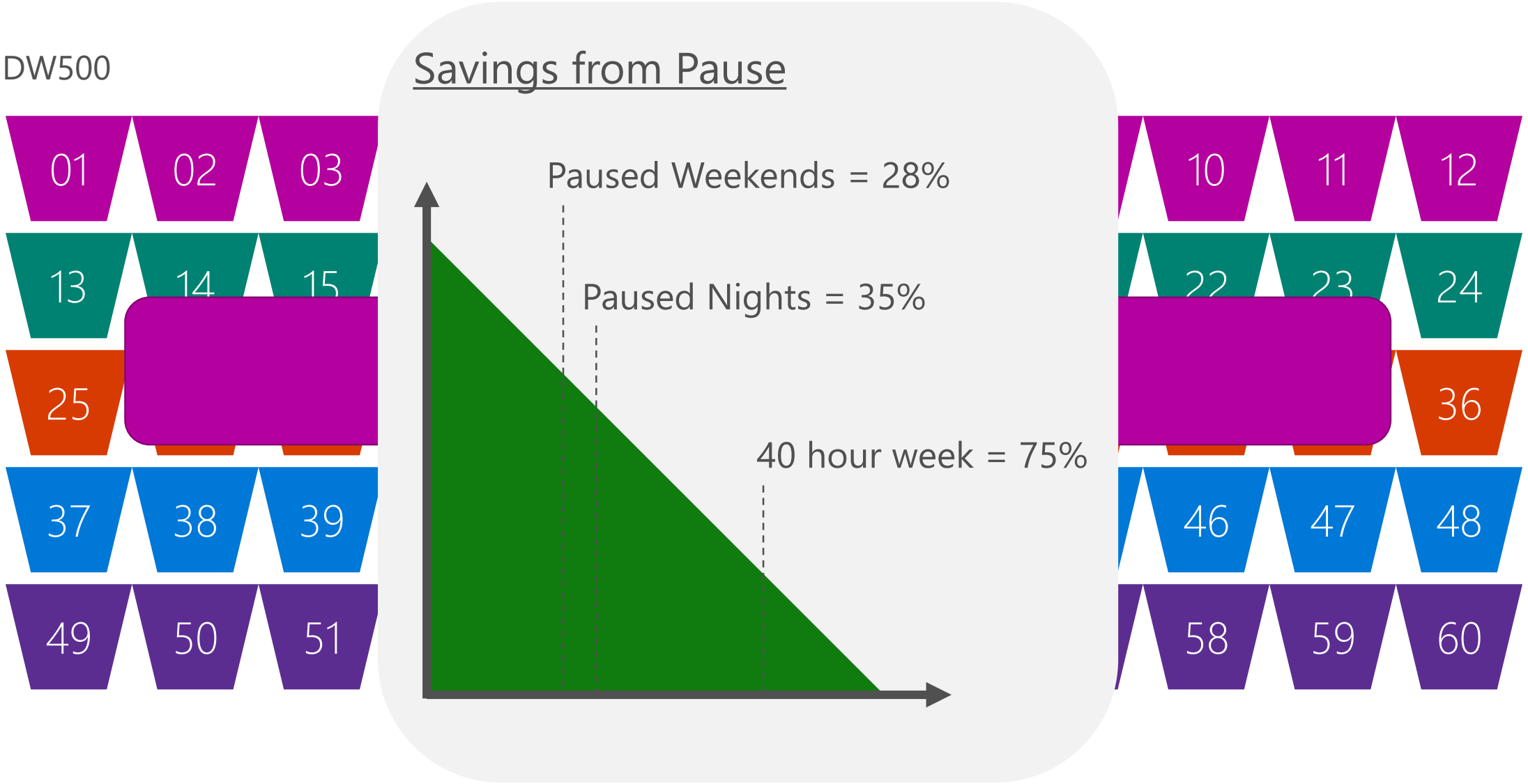


Mapping Compute in SQLDW

DW500

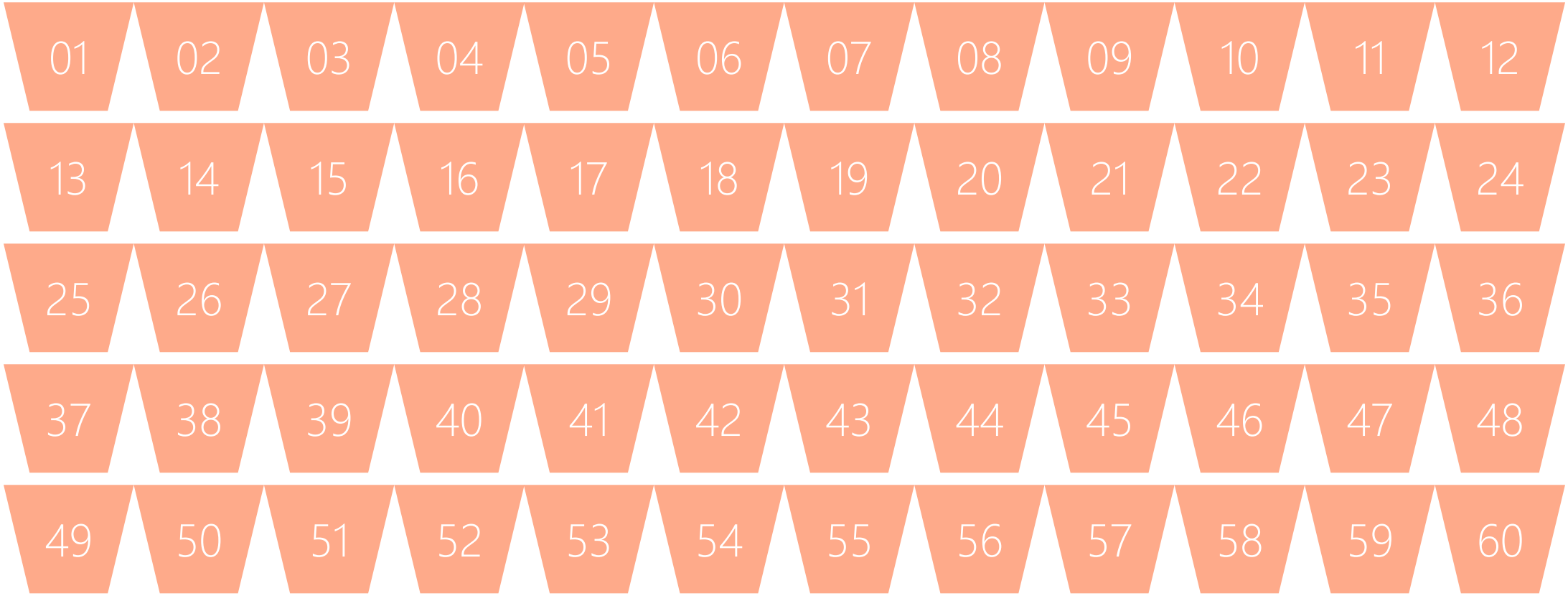


Pausing compute in SQLDW



Resuming compute in SQLDW

DW500



Simple Distributed Query

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales];
```

```
SELECT   SUM(*)
FROM     dbo.[FactInternetSales];
```

Control

Compute

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
;
```

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
;
```

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
;
```

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
;
```

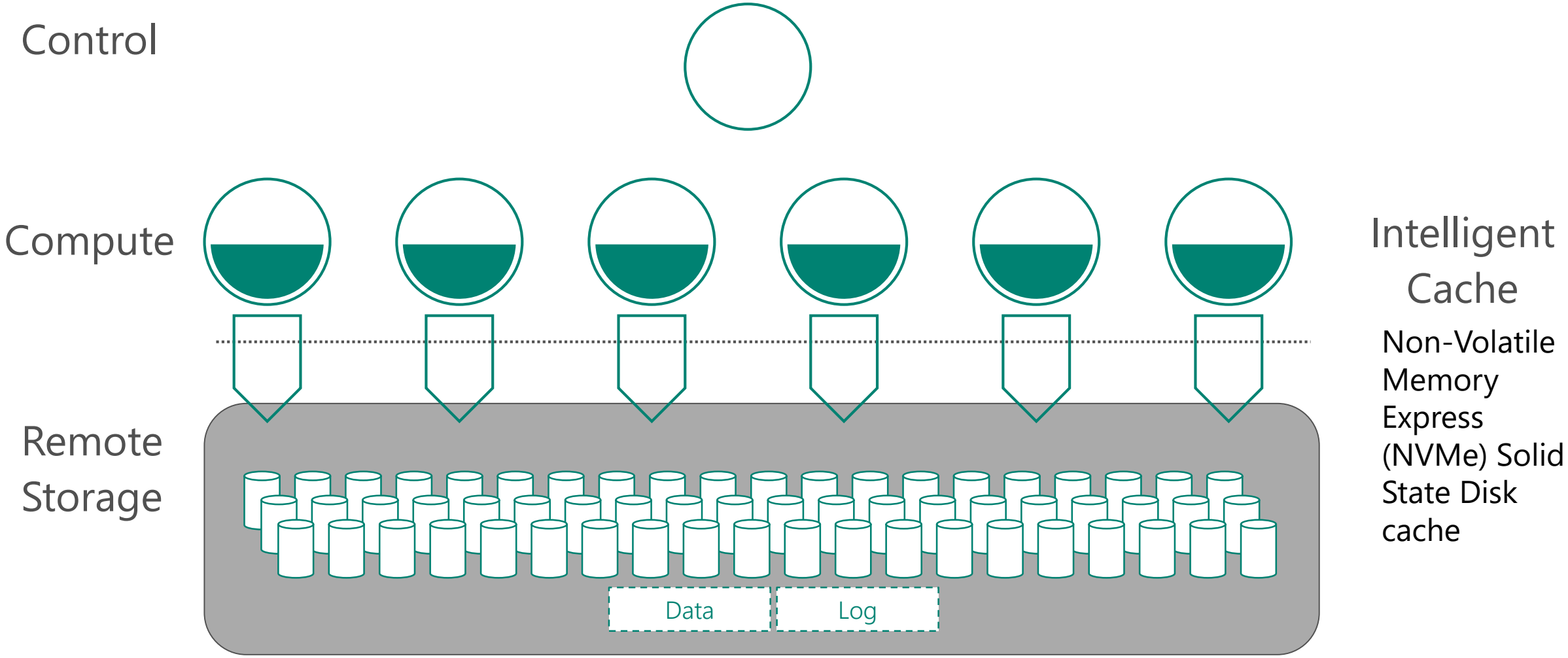
Optimized for Elasticity

Service level	Max concurrent queries	Compute nodes	Distributions per Compute node	Max memory per distribution (MB)	Max memory per data warehouse (GB)
DW100	4	1	60	400	24
DW200	8	2	30	800	48
DW300	12	3	20	1,200	72
DW400	16	4	15	1,600	96
DW500	20	5	12	2,000	120
DW600	24	6	10	2,400	144
DW1000	32	10	6	4,000	240
DW1200	32	12	5	4,800	288
DW1500	32	15	4	6,000	360
DW2000	32	20	3	8,000	480
DW3000	32	30	2	12,000	720
DW6000	32	60	1	24,000	1440

Optimized for Compute

Service level	Max concurrent queries	Compute nodes	Distributions per Compute node	Max memory per distribution (GB)	Max memory per data warehouse (GB)
DW1000c	32	2	30	10	600
DW1500c	32	3	20	15	900
DW2000c	32	4	15	20	1200
DW2500c	32	5	12	25	1500
DW3000c	32	6	10	30	1800
DW5000c	32	10	6	50	3000
DW6000c	32	12	5	60	3600
DW7500c	32	15	4	75	4500
DW10000c	32	20	3	100	6000
DW15000c	32	30	2	150	9000
DW30000c	32	60	1	300	18000

Optimised for Compute



Intelligent Cache

Non-Volatile Memory Express (NVMe) Solid State Disk cache

Intelligent Cache (Optimized for compute)

2.5x more memory on nodes

Faster cores

Local NVMe SSD locally on node

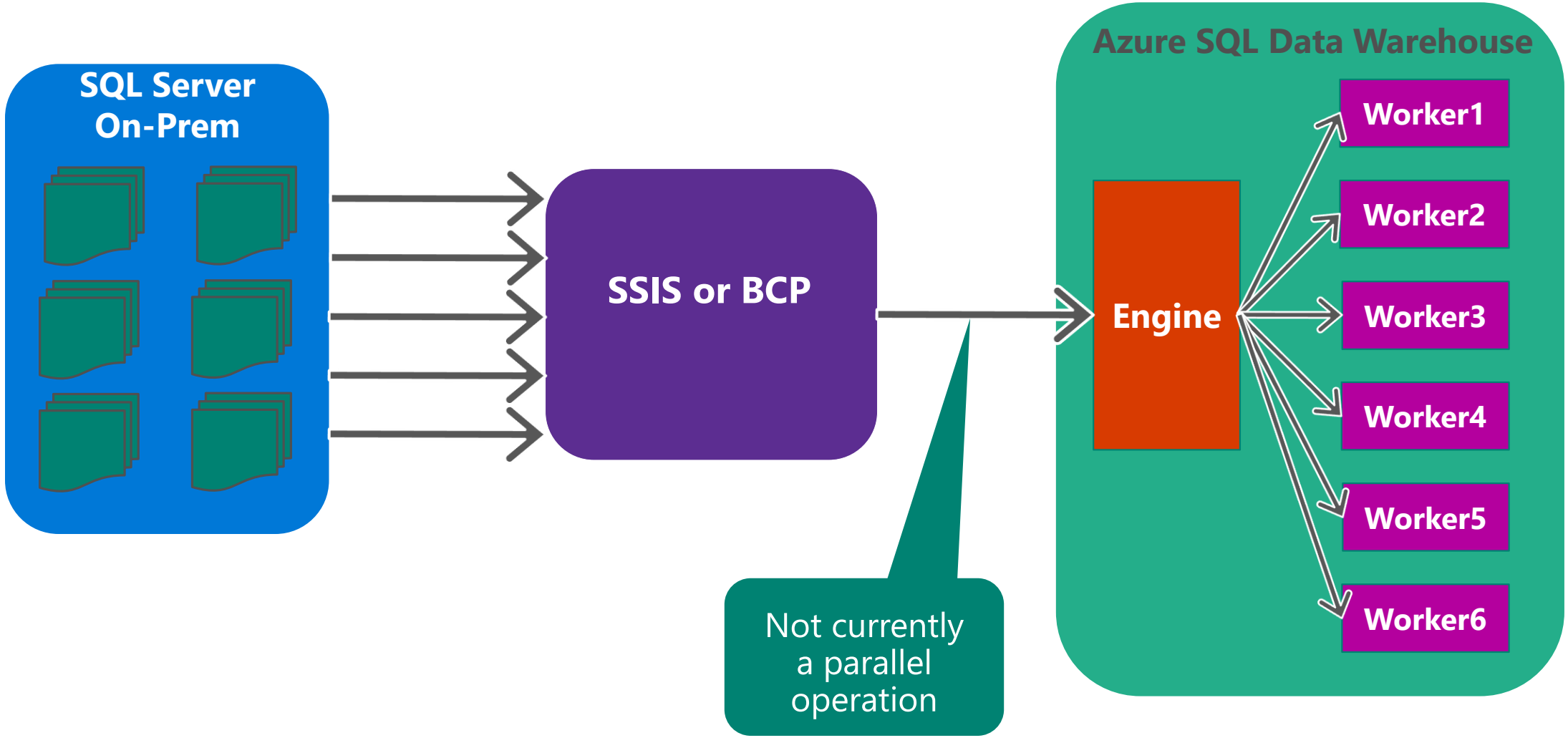
Columnstore segments can be stored in the cache so closer to compute.

- Improvements can go up to 100x

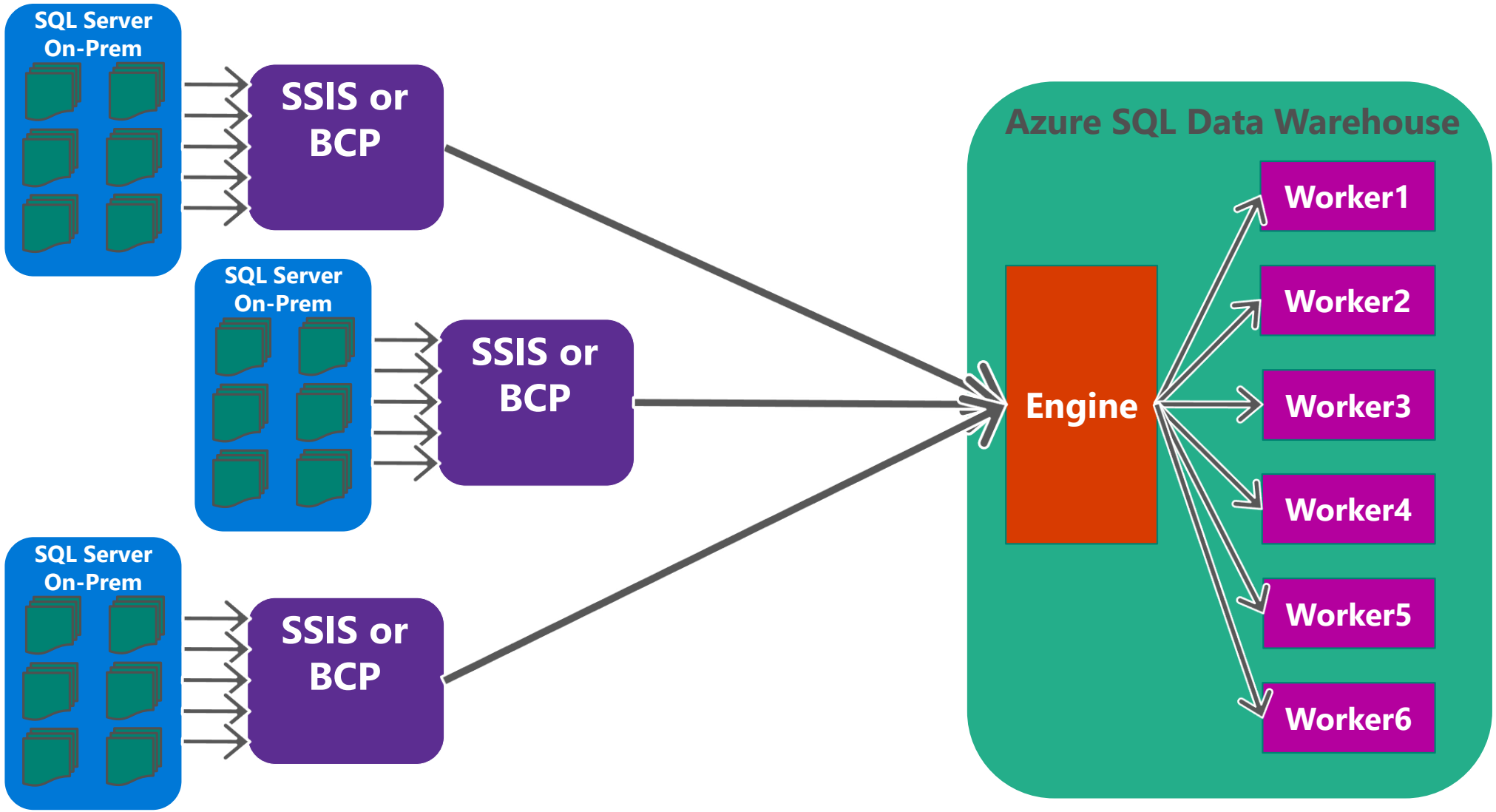
Columnstore data has now infinite size (previously caped 1.2 PB)

Loading data

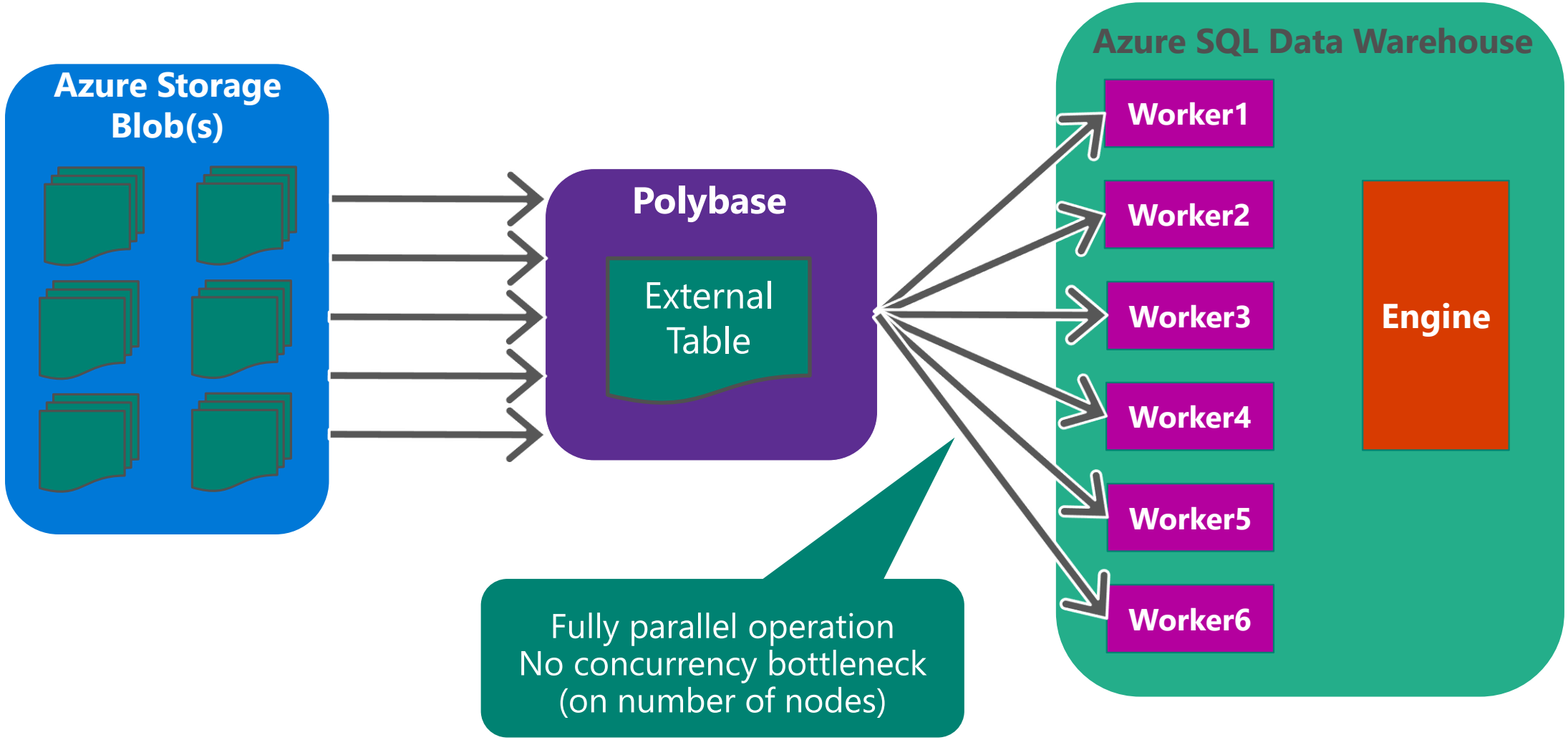
Architecture for Loading – SSIS or BCP



Parallel Loading – SSIS or BCP



Architecture for Loading - Polybase



Data loading

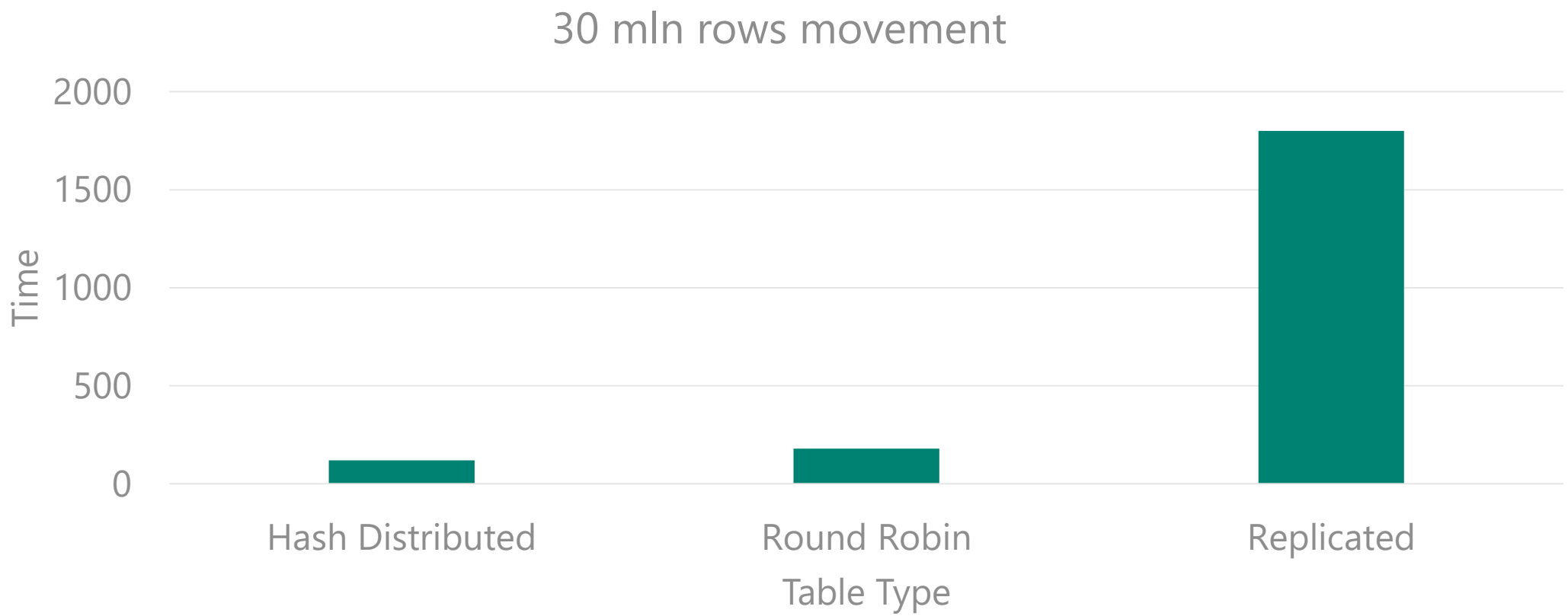
Use mediumrc+ for high DWU loads

DWU	Max External Readers	Max Wr
DW100	8	60
DW200	16	60
DW300	24	60
DW400	32	60
DW500	40	60
DW600	48	60
DW1000	80	80
DW1200	96	96
DW1500	120	120
DW2000	160	160
DW3000	240	240
DW6000	480	480

Exception - If target table is clustered index or non clustered index maximum writers is 60

Loading Data - Table Design Choices

Moving 30 mln rows into three different table distribution types



Loading Data - Index Design Choices

Moving 30 mln rows into three different index types

Tables

Table Distribution Options

Selecting the right distribution method is key to good performance

Hash Distributed	Round Robin (Default on SQL DW)	Replicated (Default on PDW)
Data divided across nodes based on hashing algorithm Same value will always hash to same distribution Optimal for large fact tables	Data distributed evenly across nodes Easy place to start, don't need to know anything about the data Useful for large tables without a good hash column	Data repeated on every node of the appliance Simplifies many query plans and reduces data movement Best for small lookup tables
Data Skew can be an issue when distributing on high frequency values which represent a large percentage of rows (e.g. NULL)	Will incur more data movement at query time Unable to perform PDW dwloader UPSERTs	Consumes 60X the space (SQL DW) Slower DML Joining two Replicated Table runs on one node

Hash Tables

Distributed Table DDL

```
CREATE TABLE aw.FactFinance(
      FinanceKey int NOT NULL,
      DateKey int NOT NULL,
      OrganizationKey int NOT NULL,
      DepartmentGroupKey int NOT NULL,
      ScenarioKey int NOT NULL,
      AccountKey int NOT NULL,
      Amount float NOT NULL)
WITH (DISTRIBUTION = HASH(FinanceKey));
```

Hash Distributed Example

Data File

ID	Name	City	Balance
201	Bob	Madison	$3,000
602	Larry	Palo Alto	$1,001
752	Anne	L.A.	$22,000
50	Liz	NYC	$2,200
86	Bob	Chicago	$180
633	Bob	London	$994
19	George	Paris	$3,105
320	Jeff	Madison	$0

Hash_Function (201) ➔ Dist_DB_2

Hash_Function (105) ➔ Dist_DB_2

Hash_Function (933) ➔ Dist_DB_4

DMS Hashing

ID	Name	...
150	George	...
220	Sally	...
50	Liz	...
320	Jeff	...

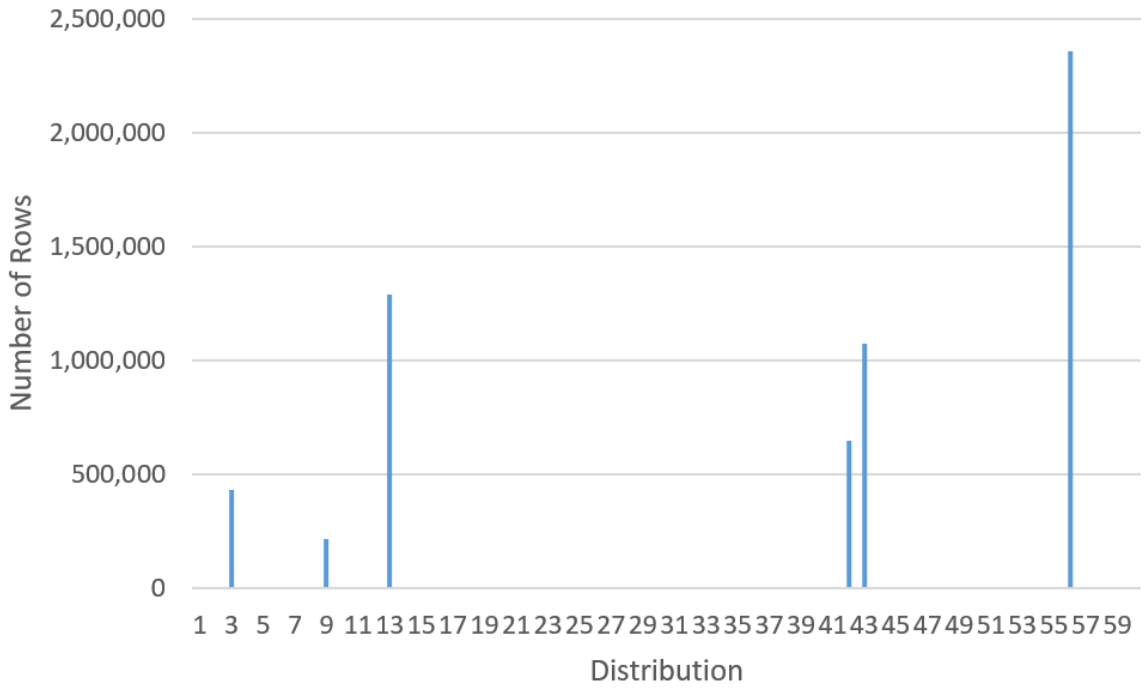
ID	Name	...
201	Bob	...
105	Sue	...
86	Bob	...

ID	Name	...
602	Larry	...
752	Anne	...

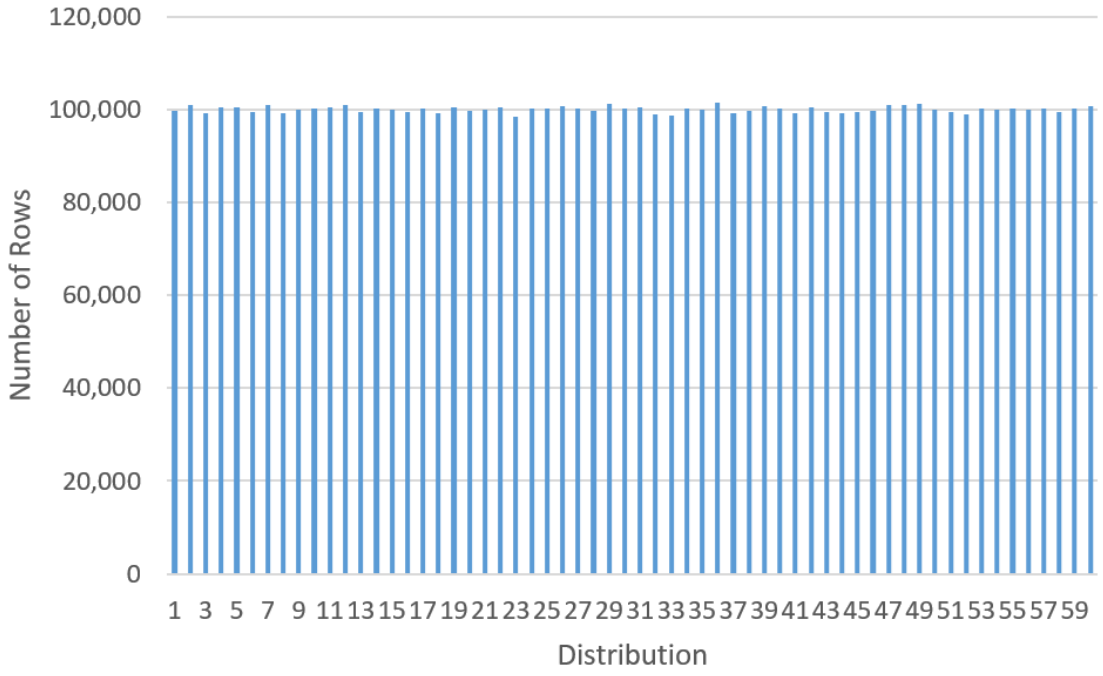
ID	Name	...
933	Mary	...
633	Bob	...
19	George	...

Watch out for Data Skew

Skewed Data



Evenly Distributed Data



Demo

Round Robin Tables

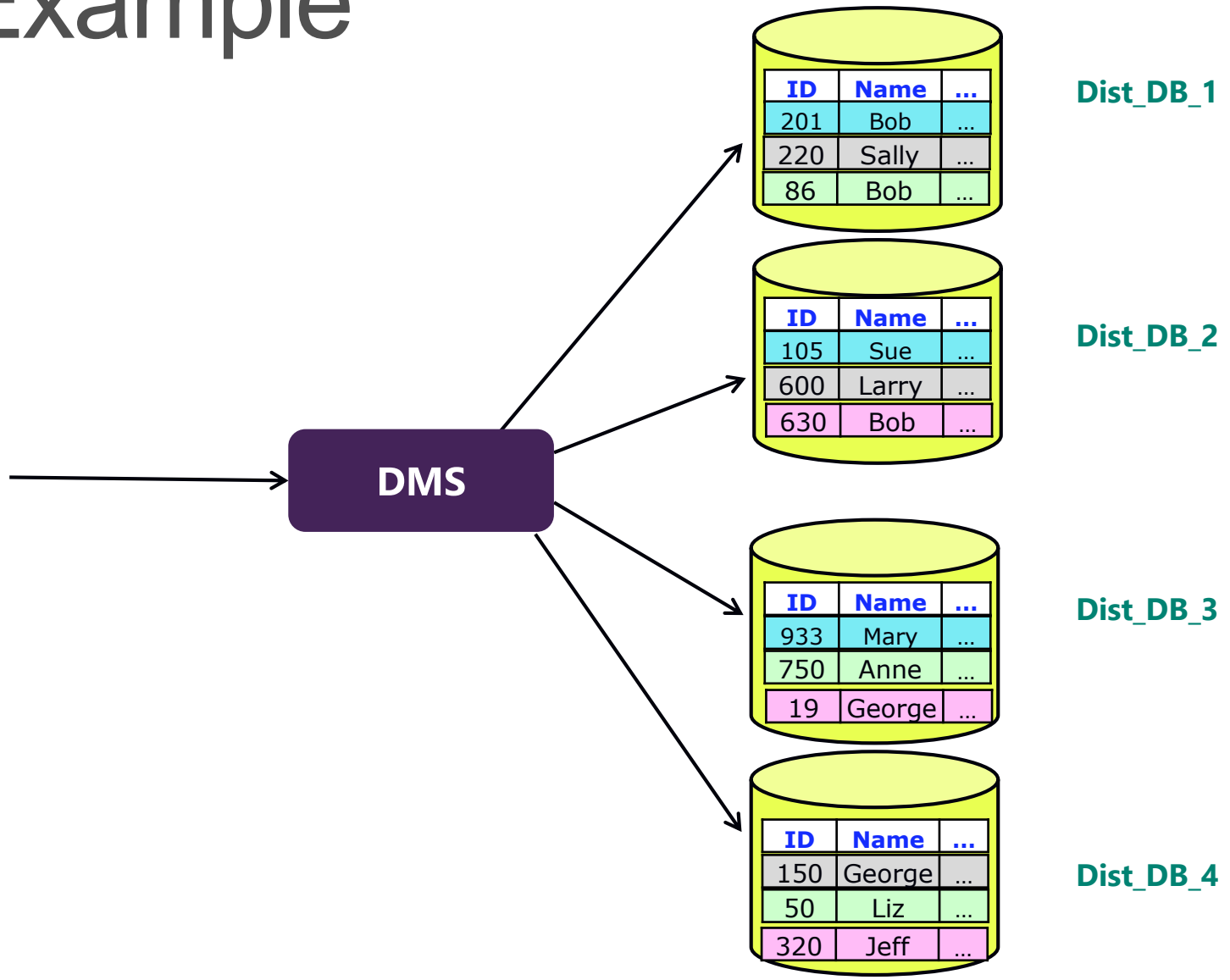
Round Robin Table DDL

```
CREATE TABLE aw.FactFinance(
      FinanceKey int NOT NULL,
      DateKey int NOT NULL,
      OrganizationKey int NOT NULL,
      DepartmentGroupKey int NOT NULL,
      ScenarioKey int NOT NULL,
      AccountKey int NOT NULL,
      Amount float NOT NULL)
WITH (DISTRIBUTION = ROUND_ROBIN);
```

Round Robin Example

Data File

ID	Name	City	Balance
201	Bob	Madison	$3,000
105	Sue	San Fran	$110
933	Mary	Seattle	$40,000
150	George	Seattle	$60
220	Sally	Mtn View	$990
600	Larry	Palo Alto	$1,001
750	Anne	L.A.	$22,000
50	Liz	NYC	$2,200
86	Bob	Chicago	$180
630	Bob	London	$994
19	George	Paris	$3,105
320	Jeff	Madison	$0



Replicated Tables

Replicated Table DDL

```
CREATE TABLE aw.FactFinance(
      FinanceKey int NOT NULL,
      DateKey int NOT NULL,
      OrganizationKey int NOT NULL,
      DepartmentGroupKey int NOT NULL,
      ScenarioKey int NOT NULL,
      AccountKey int NOT NULL,
      Amount float NOT NULL)
WITH (DISTRIBUTION = REPLICATE);
```

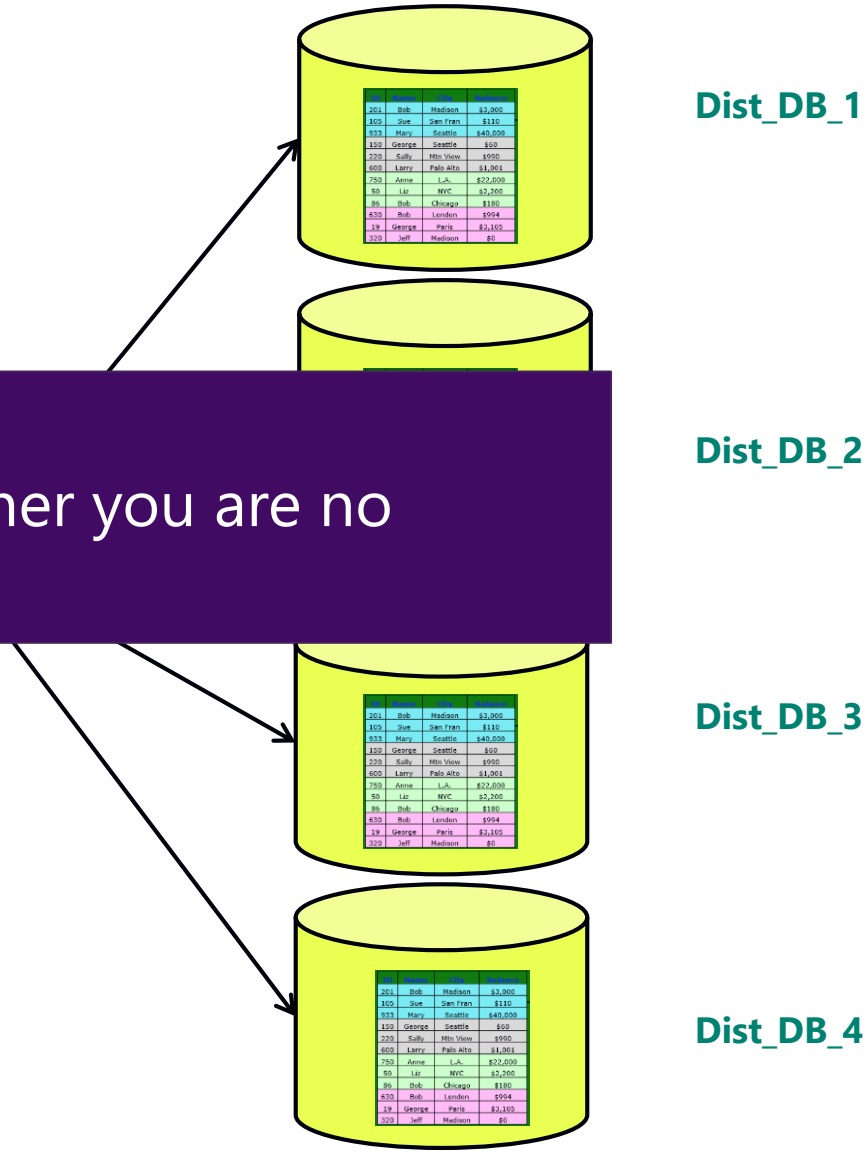
Replicated Example

PDW Only Feature
Coming to SQL DW Soon

Data File

ID			
201			
105			
933			
150	George	Seattle	$60
220	Sally	Mtn View	$990
600	Larry	Palo Alto	$1,001
750	Anne	L.A.	$22,000
50	Liz	NYC	$2,200
86	Bob	Chicago	$180
630	Bob	London	$994
19	George	Paris	$3,105
320	Jeff	Madison	$0

Tip:
If you join two replicated tables together you are no longer using an MPP system



Dist_DB_1

Dist_DB_2

Dist_DB_3

Dist_DB_4

Partitioning Tables

DDL Example of Partitioning

```
CREATE TABLE aw.FactFinance(
      FinanceKey int NOT NULL,
      DateKey int NOT NULL,
      OrganizationKey int NOT NULL,
      DepartmentGroupKey int NOT NULL,
      ScenarioKey int NOT NULL,
      AccountKey int NOT NULL,
      Amount float NOT NULL)
WITH (DISTRIBUTION = HASH(FinanceKey),
      PARTITION (DateKey RANGE RIGHT FOR VALUES
            (20100101,20200101,20300101))
       );
```

Optimizing with Partitioning

Performance optimizations include…

- Typically partition on date column
- Improves performance of predicates
- Optimize load performance and data management through partition switching, merging and splitting
- Re-indexing by partition
- Too many partitions can slow things down quite a bit.
- Do not over-partition Clustered Columnstore Tables

ASDW != ASDB

Compression

- Columnstore
  - Standard column store compression (~5x)
  - Improved with DATA_COMPRESSION = COLUMNSTORE_ARCHIVE
- Rowstore
  - Page compression on by default
    - Can not be turned off

Read Uncommitted

A

- Atomicity
  - Requests will complete or fail as a single unit

C

- Consistency
  - Database will move from one legal state to another

I

- Isolation
  - Requests will be isolated from each other

D

- Durability
  - Once a transaction has committed it will remain so

Transactions

- Transaction Limitations
  - No distributed transactions
  - No nested transactions permitted
  - No save points allowed
  - No DDL (Create Table etc) inside a transaction
- Avoid long running transactions
- Read uncommitted transactions

HA DR

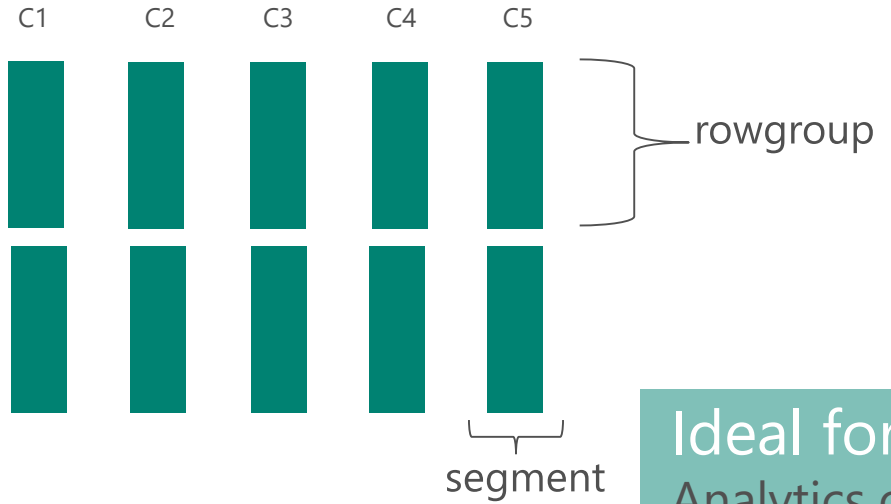
- As a benefit of using Azure Premium Storage, SQL Data Warehouse uses Azure Storage Blob snapshots to backup the primary data warehouse.
- For high availability dual load the data into two ASDW
- SLA 99.9% (43 minutes a month)

Columnstore

Columnstore Index

Data stored as rows

Data stored as columns



Ideal for OLTP

Frequent read/write on small set of rows

Ideal for Data Warehouse

Analytics on large number of rows

Improved compression:
Data from same domain compress better

Reduced I/O:
Fetch only columns needed

Improved Performance:
More data fits in memory
Optimized for CPU utilization

Batch Mode Execution
Vector Processing

Columnstore

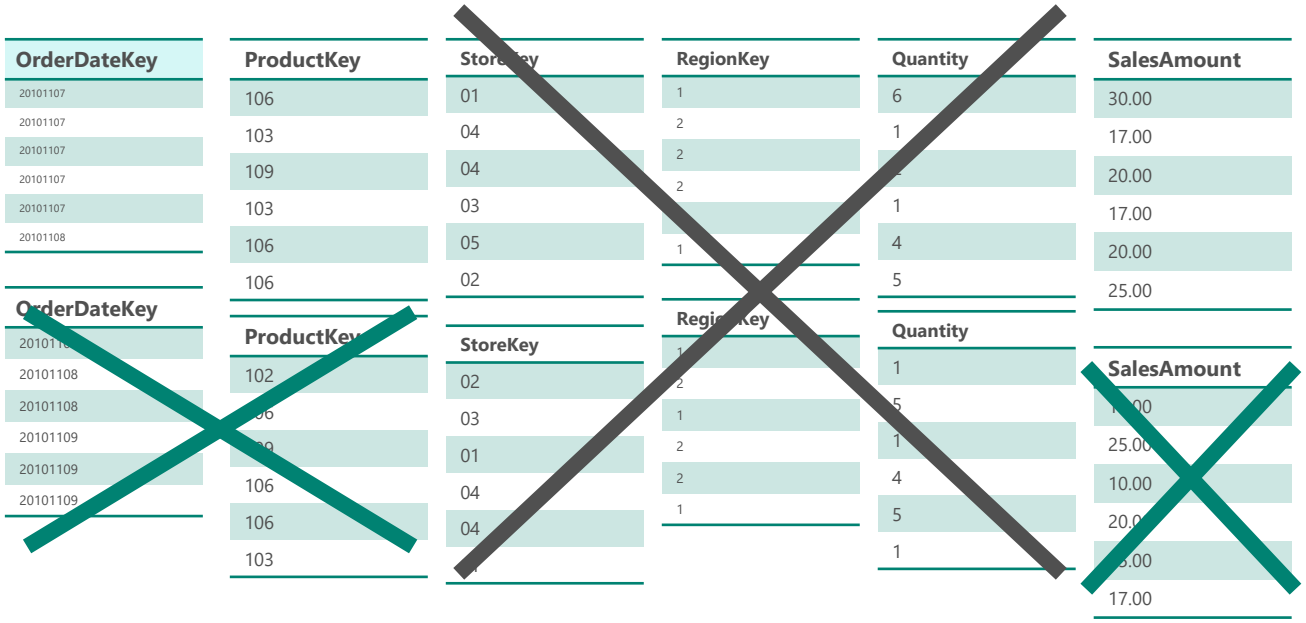
- Performance Demo

Why use column store?

- Query Performance
  - Several orders of magnitude greater
- Data Compression
  - Less space on disk and in memory
- Low index maintenance / optimisations / designs
  - Only one index required

Query Processing - Read The Data Needed

SELECT **ProductKey**, SUM (**SalesAmount**)
FROM SalesTable
WHERE **OrderDateKey** < 20101108



Column Elimination

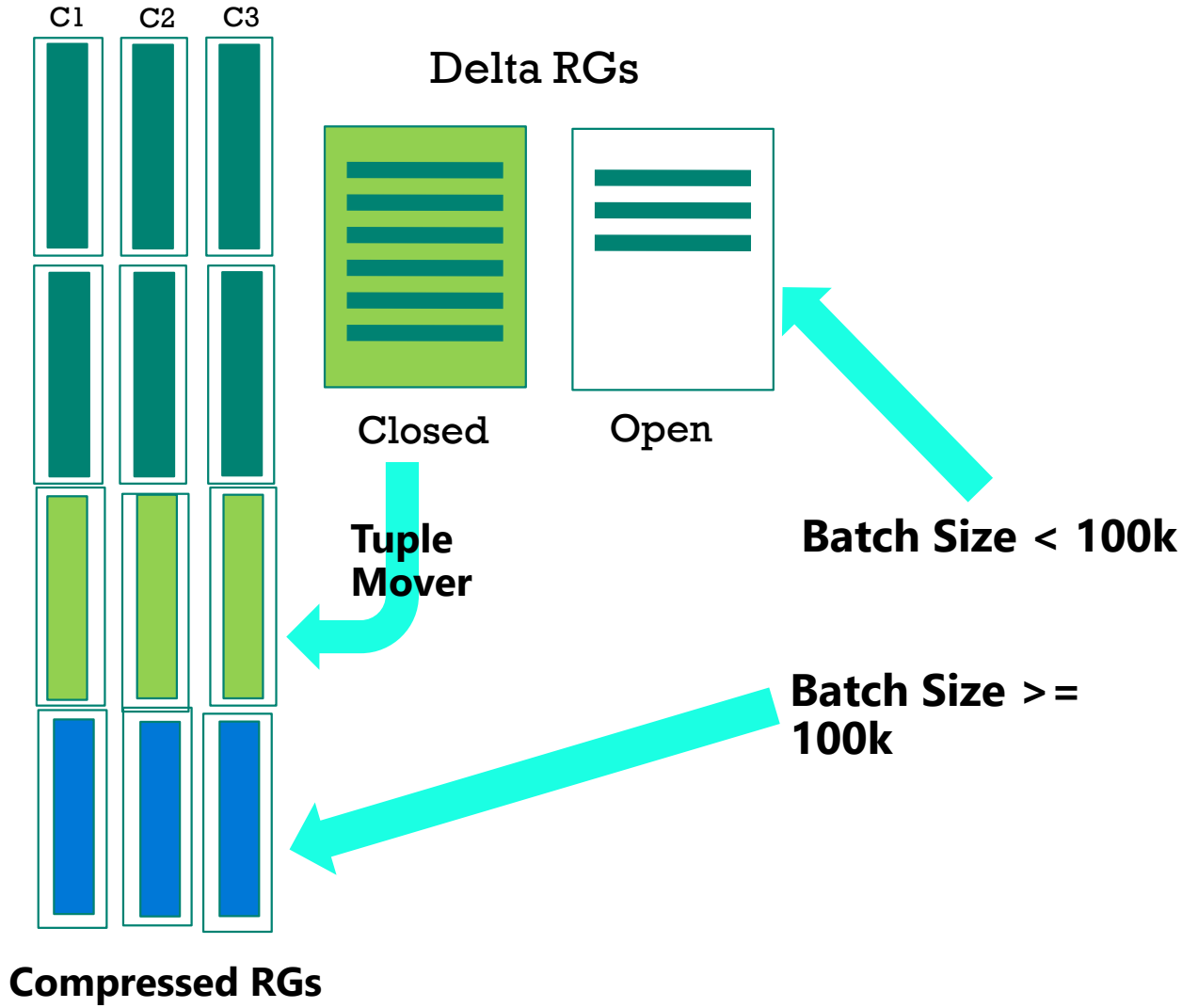
Segment Elimination

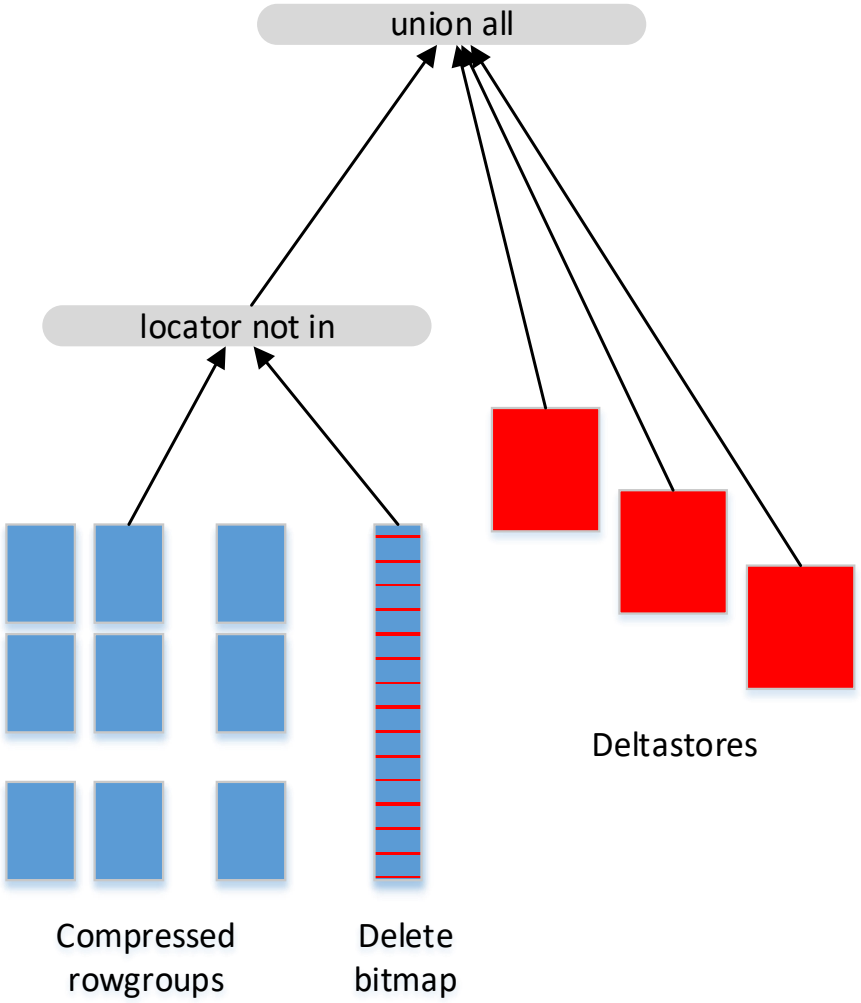
Columnstore health

- 1 million is good
- >100,000 is acceptable
- <100,000 is bad
- Updating/Deleting/Inserting = deltastore/rowstore partition

Batch loading

- < 102,400 will go to the rowstore
- => 102,400 will go straight to columnstore
- 60 distributions
  - 60 * 102,400 = 6,144,000
- 60 distributions with 4 partitions
  - 60 * 4 * 102,400 = 24,576,000
- 102,400 will stay as compressed segments
- Less that this will build up into CLOSED partitions (1048576 rows)
  - Tuple Mover will come along and move to COMPRESSED
  - Separate dictionary object
- Demo





Data batch load

Tuple Mover kicking in to compress the data as it loads

Columnstore in Azure SQL Datawarehouse

- sys.pdw_nodes_column_store_row_groups
  - State of various row groups (open, closed, compressed)
  - Total rows
- sys.pdw_nodes_column_store_segments
  - Encoding used
  - Dictionary id
  - Min,max metadata
- sys.pdw_nodes_column_store_dictionaries
  - On disk dictionary size
- sys.dm_pdw_nodes_db_column_store_row_group_physical_stats
  - Current rowgroup-level information about all of the columnstore indexes
- Maintenance Demo

Columnstore in Azure SQL Datawarehouse

- SELECT * FROM sys.dm_pdw_nodes_db_column_store_row_group_physical_stats ORDER BY trim_reason_desc DESC

1 - NO_TRIM: The row group was not trimmed. The row group was compressed with the maximum of 1,048,476 rows. The number of rows could be less if a subsset of rows was deleted after delta rowgroup was closed

2 – BULKLOAD: The bulk load batch size limited the number of rows.

3 – REORG: Forced compression as part of REORG command.

4 – DICTIONARY_SIZE: Dictionary size grew too big to compress all of the rows together.

5 – MEMORY_LIMITATION: Not enough available memory to compress all the rows together.

6 – RESIDUAL_ROW_GROUP: Closed as part of last row group with rows < 1 million during index build operation

Performance

SQL DW Workload

SQL DW is designed for DW and not OLTP. All the traditional DW workload characteristics apply

- Not good for singleton DML heavy operations, Example: Clients issuing singleton update, insert, delete (Not for chatty workload)

Incremental data is loaded regularly by ETL/ELT process in batch mode. Not optimal for real time ingestion

DW workload typically considers to be tier-2 SLA

Low number of concurrent queries

Concurrency

- Concurrency queries
  - 1,024 concurrent connections
  - Max 32 concurrent queries at the same time
  - A query will take up 1 or more concurrency slots
    - Based on
      - DWU level of the ASDW
      - Resource class
- Concurrency slots
  - Increased with DWU
  - If max concurrent queries or max concurrency slots are reached query will be queued

Concurrency Slot Consumption	DW 100	DW 200	DW 300	DW 400	DW 500	DW 600	DW 1000	DW 1200	DW 1500	DW 2000
Max Concurrent Queries	32	32	32	32	32	32	32	32	32	32
Max Concurrency Slots	4	8	12	16	20	24	40	48	60	80

Concurrency

- Workload Management
- Roles
  - Smallrc (By default, each user is a member of the small resource class, smallrc)
  - mediumrc
  - largerc
  - xlargerc

DWU	Maximum concurrent queries	Concurrency slots allocated	Slots used by smallrc	Slots used by mediumrc	Slots used by largerc	Slots used by xlargerc
DW100	4	4	1	1	2	4
DW200	8	8	1	2	4	8
DW300	12	12	1	2	4	8
DW400	16	16	1	4	8	16
DW500	20	20	1	4	8	16
DW600	24	24	1	4	8	16
DW1000	32	40	1	8	16	32
DW1200	32	48	1	8	16	32
DW1500	32	60	1	8	16	32
DW2000	32	80	1	16	32	64
DW3000	32	120	1	16	32	64
DW6000	32	240	1	32	64	128

Static Resource Classes

- Static memory assignment and concurrency slots
- Better control over user resource usage when scaling
- Increase concurrency
- More granularity over resource consumption

Static Resource Classes

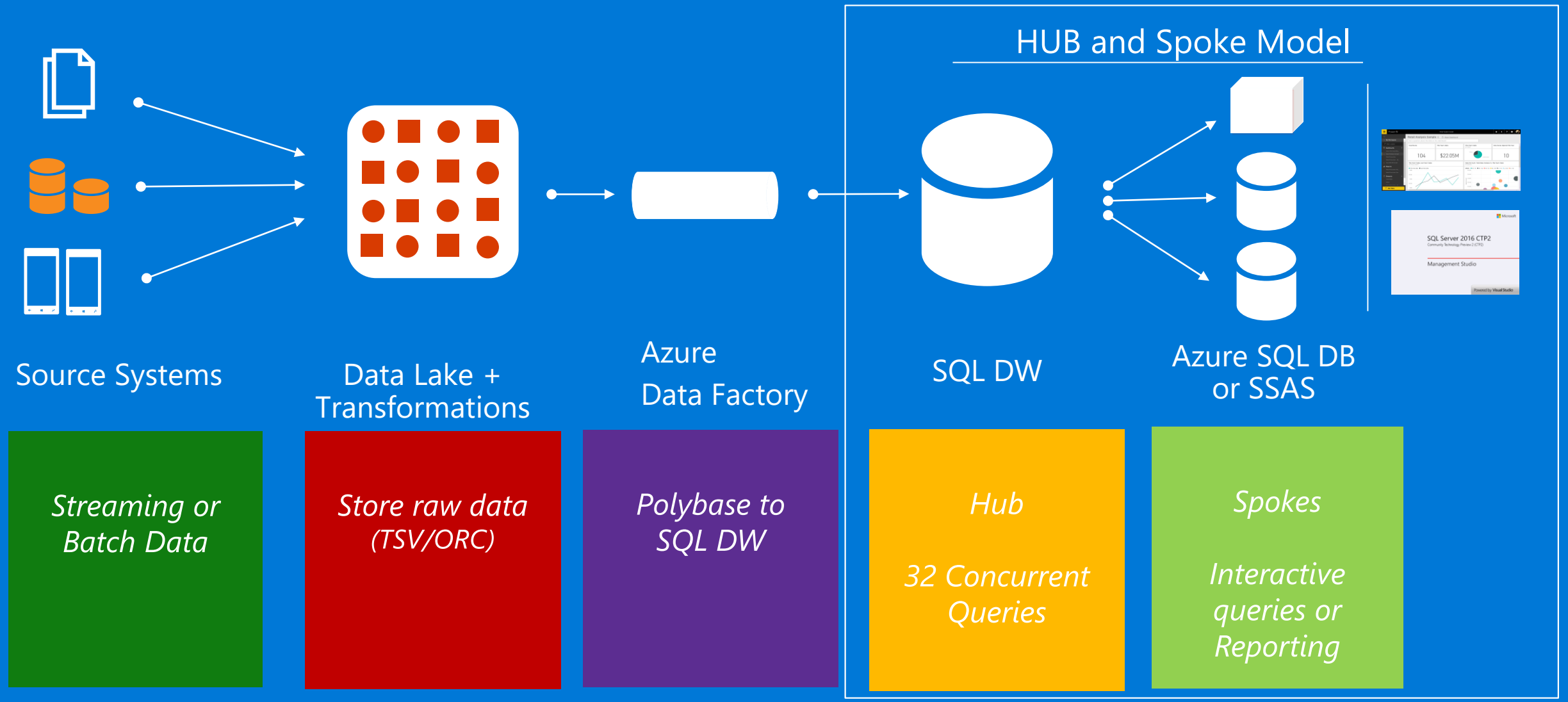
DWU	Maximum concurrent queries	Concurrency slots allocated	staticrc10	staticrc20	staticrc30	staticrc40	staticrc50	staticrc60	staticrc70	staticrc80
DW100	4	4	1	2	4	4	4	4	4	4
DW200	8	8	1	2	4	8	8	8	8	8
DW300	12	12	1	2	4	8	8	8	8	8
DW400	16	16	1	2	4	8	16	16	16	16
DW500	20	20	1	2	4	8	16	16	16	16
DW600	24	24	1	2	4	8	16	16	16	16
DW1000	32	40	1	2	4	8	16	32	32	32
DW1200	32	48	1	2	4	8	16	32	32	32
DW1500	32	60	1	2	4	8	16	32	32	32
DW2000	32	80	1	2	4	8	16	32	64	64
DW3000	32	120	1	2	4	8	16	32	64	64
DW6000	32	240	1	2	4	8	16	32	64	128

Resource Classes

DWU	smallrc	mediumrc	largerc	xlargerc
DW100	100	100	200	400
DW200	100	200	400	800
DW300	100	200	400	800
DW400	100	400	800	1,600
DW500	100	400	800	1,600
DW600	100	400	800	1,600
DW1000	100	800	1,600	3,200
DW1200	100	800	1,600	3,200
DW1500	100	800	1,600	3,200
DW2000	100	1,600	3,200	6,400
DW3000	100	1,600	3,200	6,400
DW6000	100	3,200	6,400	12,800

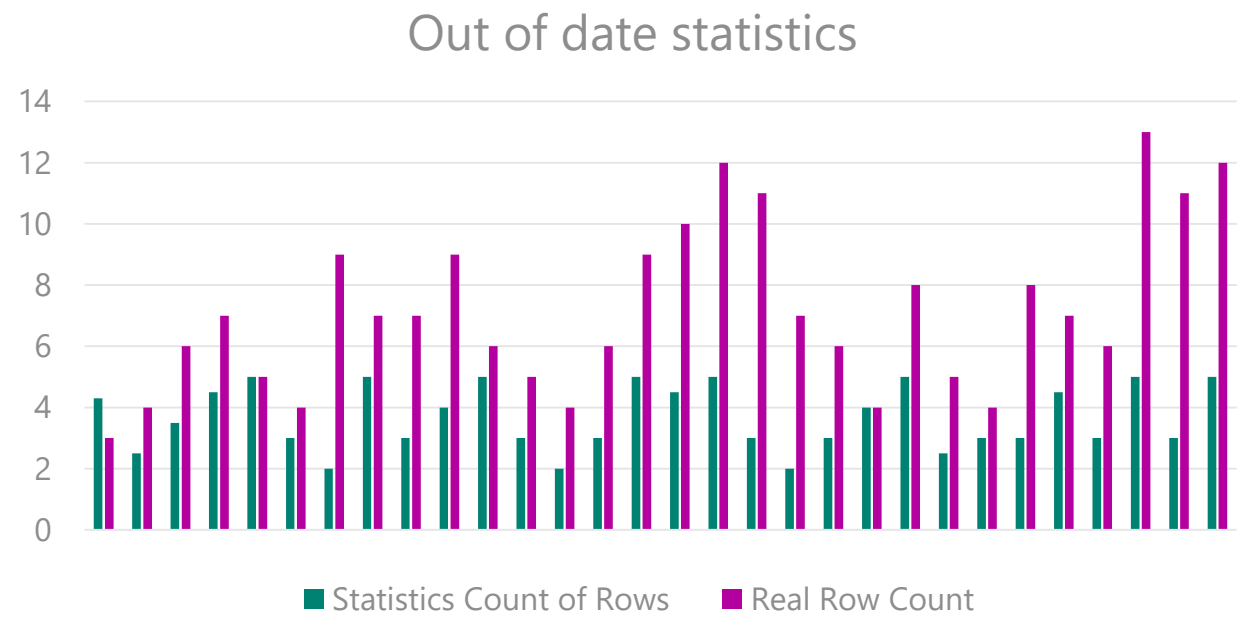
DWU	staticrc10	staticrc20	staticrc30	staticrc40	staticrc50	staticrc60	staticrc70	staticrc80
DW100	100	200	400	400	400	400	400	400
DW200	100	200	400	800	800	800	800	800
DW300	100	200	400	800	800	800	800	800
DW400	100	200	400	800	1,600	1,600	1,600	1,600
DW500	100	200	400	800	1,600	1,600	1,600	1,600
DW600	100	200	400	800	1,600	1,600	1,600	1,600
DW1000	100	200	400	800	1,600	3,200	3,200	3,200
DW1200	100	200	400	800	1,600	3,200	3,200	3,200
DW1500	100	200	400	800	1,600	3,200	3,200	3,200
DW2000	100	200	400	800	1,600	3,200	6,400	6,400
DW3000	100	200	400	800	1,600	3,200	6,400	6,400
DW6000	100	200	400	800	1,600	3,200	6,400	12,800

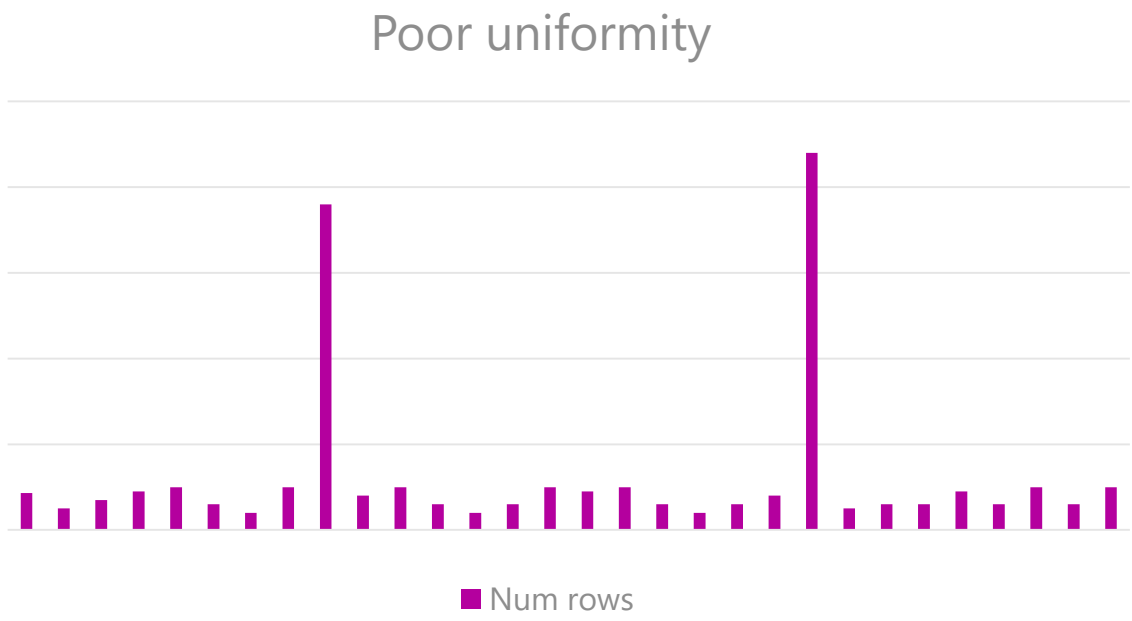
Hub and Spoke Model



Statistics

- ASDW does not automatically create or update statistics
  - Create statistics on each column?
  - Create statistics on multi column?

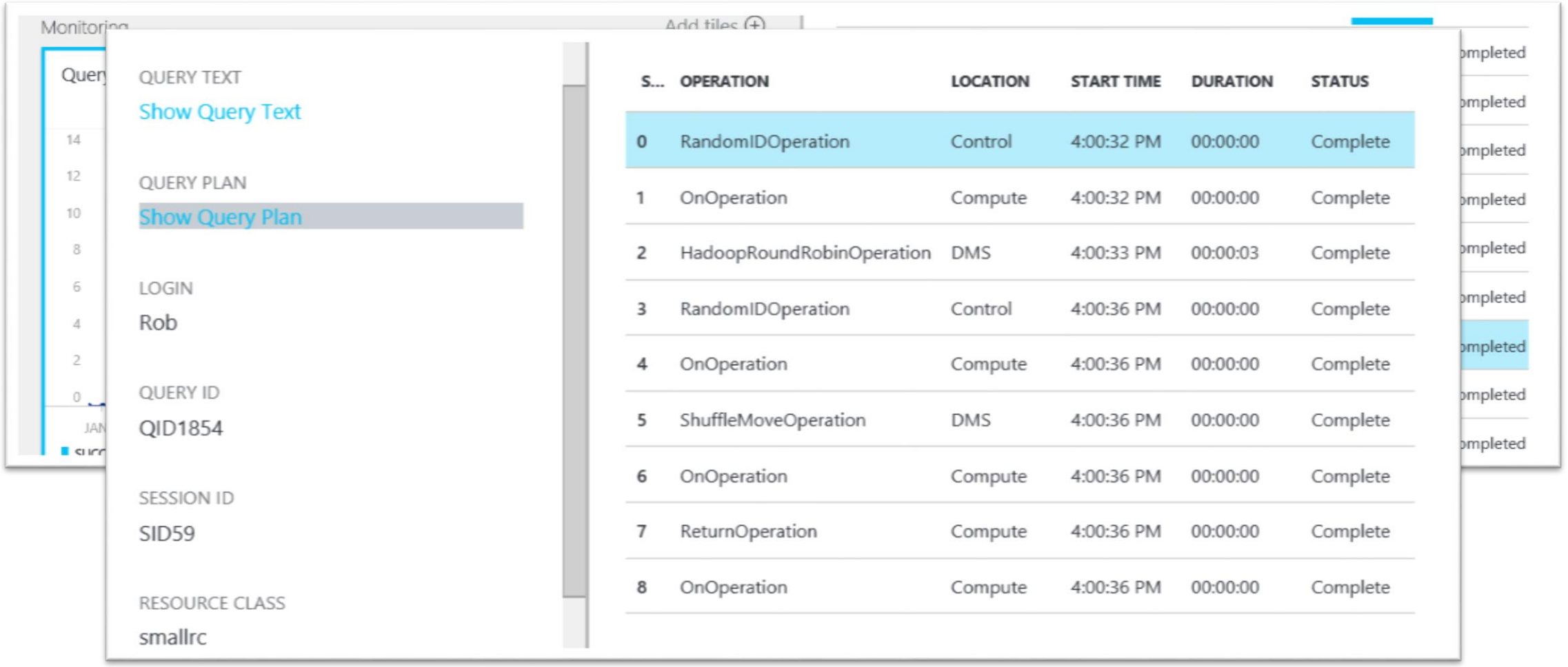




Statistics

- But isn't SQL DW just lots of SQL database. Why have statistics been turned off?
- Demo
- Poor statistics can lead to
  - Executing the wrong data movement operation
    - Data movement on larger fact tables rather than dimension tables
  - Executing the wrong data movement type

Jobs and Query Plans

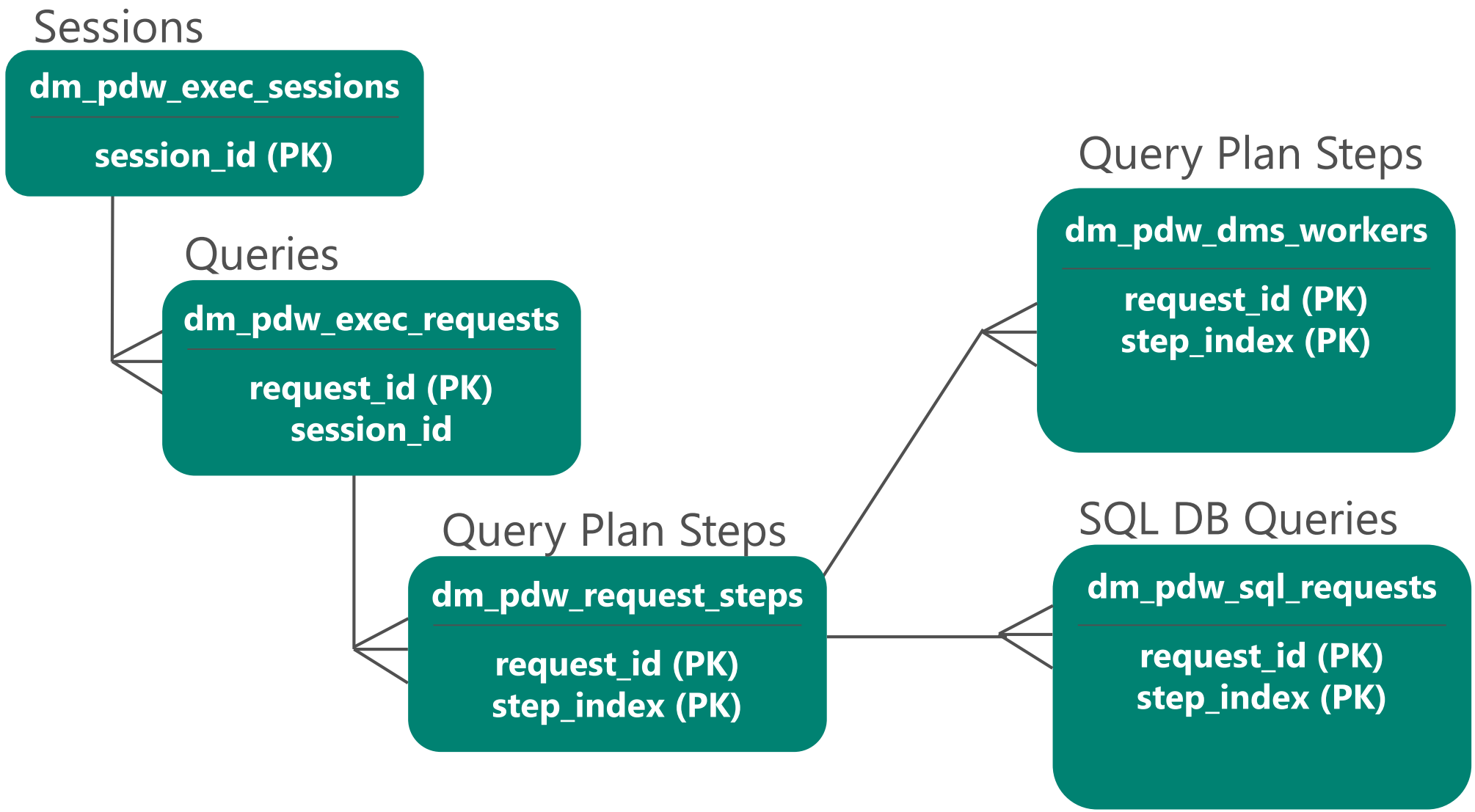


Jobs and Query Plans

- DMVs and viewing query plans
  - Demo

```
<?xml version="1.0" encoding="utf-8"?>
<dsql_query number_nodes="2" number_distributions="60" number_distributions_per_node="30">
  <sql>SELECT Crimetype, COUNT(Crimetype) FROM dbo.PoliceData GROUP BY Crimetype</sql>
  <dsql_operations total_cost="0.00632269632" total_number_operations="5">
    <dsql_operation operation_type="RND_ID">
      <identifier>TEMP_ID_342</identifier>
    </dsql_operation>
    <dsql_operation operation_type="ON">
      <location permanent="false" distribution="AllDistributions" />
      <sql_operations>
        <sql_operation type="statement">CREATE TABLE [tempdb].[dbo].[TEMP_ID_342] ([Crimetype] VARCHAR(255) COL
      </sql_operations>
    </dsql_operation>
    <dsql_operation operation_type="SHUFFLE_MOVE">
      <operation_cost cost="0.00632269632" accumulative_cost="0.00632269632" average_rowsize="26.344568" output
      <source_statement>SELECT [T1_1].[Crimetype] AS [Crimetype],
       [T1_1].[col] AS [col]
FROM   (SELECT   COUNT_BIG([T2_1].[Crimetype]) AS [col],
                 [T2_1].[Crimetype] AS [Crimetype]
        FROM     [ASDWDemoRAL01].[dbo].[PoliceData] AS T2_1
        GROUP BY [T2_1].[Crimetype]) AS T1_1</source_statement>
      <destination_table>[TEMP_ID_342]</destination_table>
      <shuffle_columns>Crimetype;</shuffle_columns>
    </dsql_operation>
    <dsql_operation operation_type="RETURN">
      <location distribution="AllDistributions" />
      <select>SELECT [T1_1].[Crimetype] AS [Crimetype],
       [T1_1].[col] AS [col]
FROM   (SELECT CONVERT (INT, [T2_1].[col], 0) AS [col],
```

Query Execution DMVs



Investigating Performance

- Watching a plan progression
  - Demo

Investigating Performance

- View plans on distributions
  - Demo
  - (only shows the estimated execution plan from the cache)

```
--Conn1
SELECT COUNT_BIG(*) FROM PoliceData AS A
CROSS JOIN PoliceData AS B
CROSS JOIN PoliceData AS C
OPTION (LABEL = 'myquerytest')

--Conn2
SELECT
CONCAT('DBCC PDW_SHOWEXECUTIONPLAN (',distribution_id, ',', sql_spid, ')', CHAR(13), CHAR(10)) DBCCCommand
,*
FROM sys.dm_pdw_dms_workers
WHERE request_id IN (
SELECT request_id
FROM sys.dm_pdw_exec_requests
WHERE status not in ('Completed','Failed','Cancelled')
AND session_id <> session_id()
AND[label] = 'myquerytest')
AND STATUS != 'StepComplete'
AND distribution_id != -1

--Save plan as .sqlplan and open in management studio
DBCC PDW_SHOWEXECUTIONPLAN (43,1231)
```

Investigating Performance

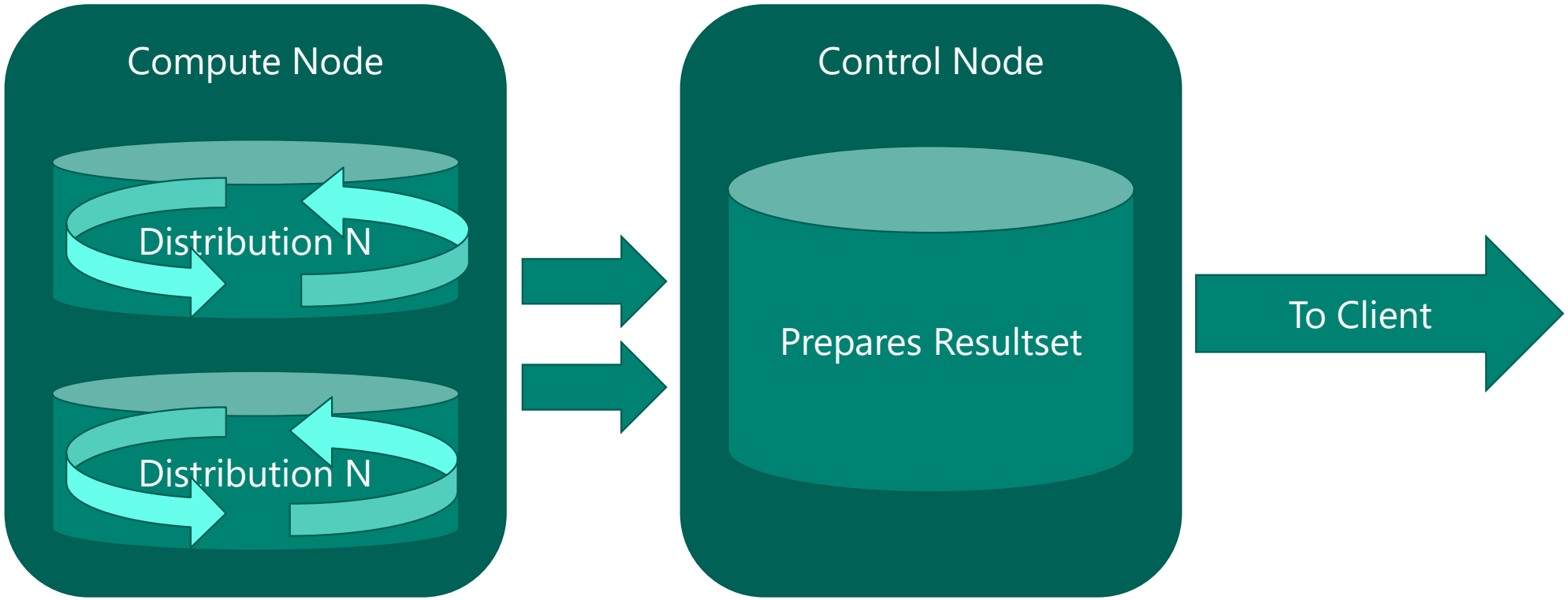
- Waiting tasks
  - Demo

```
--Conn1
BEGIN TRAN

UPDATE A
SET Longitude = 10
FROM [PoliceDataLoadingDemo] AS A
WHERE CrimeID = '9017e91cc813aec96205962f13f1e0054ba05a82c61e0a74c6258f2377ec428f'
OPTION (LABEL = 'myquerytest')

--Conn2
UPDATE A
SET Longitude = 12
FROM [PoliceDataLoadingDemo] AS A
WHERE CrimeID = '9017e91cc813aec96205962f13f1e0054ba05a82c61e0a74c6258f2377ec428f'
OPTION (LABEL = 'myquerytest')

--Conn3
SELECT waits.session_id,
      waits.request_id,
      requests.command,
      requests.status,
      requests.start_time,
      waits.type,
      waits.state,
      waits.object_type,
      waits.object_name
FROM   sys.dm_pdw_waits waits
   JOIN  sys.dm_pdw_exec_requests requests
   ON waits.request_id=requests.request_id
WHERE waits. request_id IN (SELECT request_id
FROM sys.dm_pdw_exec_requests
WHERE status not in ('Completed','Failed','Cancelled')
AND session_id <> session_id()
AND[label] = 'myquerytest')
ORDER BY waits.object_name, waits.object_type, waits.state;
```

Data Movement

- Data Movement will not be invoked when
  - Two distribution compatible tables are joined
  - A table is joined to a replicated table
  - Aggregation is distribution compatible
- Data Movement does occur when
  - Two distribution incompatible tables are joined
  - Round robin tables are distribution incompatible with all tables, except replicated tables
  - Aggregation is distribution incompatible

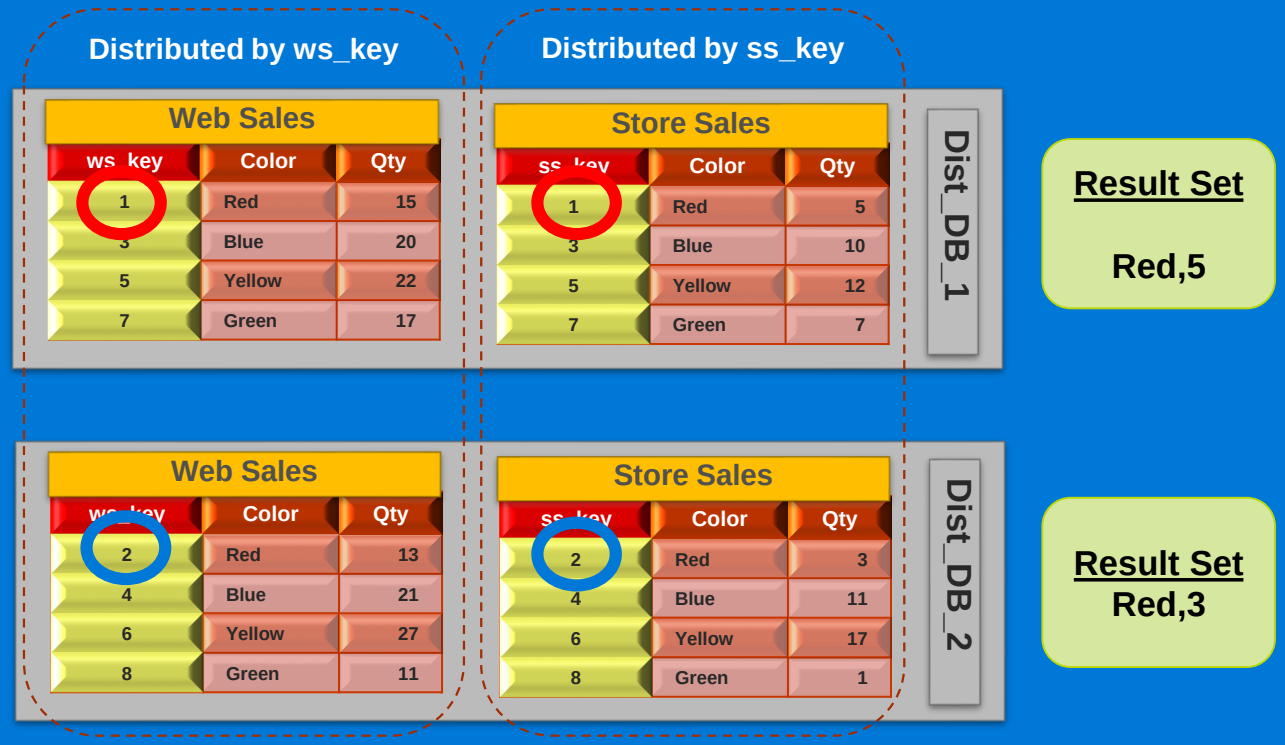
Resultset resolves within distribution



PartitionMoveOperation

Distribution Compatible Join of Two Distributed Tables

```
SELECT a.color, b.Qty
  FROM web_sales a
  JOIN store_sales b ON ws_key = ss_key
 WHERE a.color = 'Red'
```



Result Set

Red,5

Result Set
Red,3

Final Result Set

Red,5 : Red,3

Distribution Compatible Join

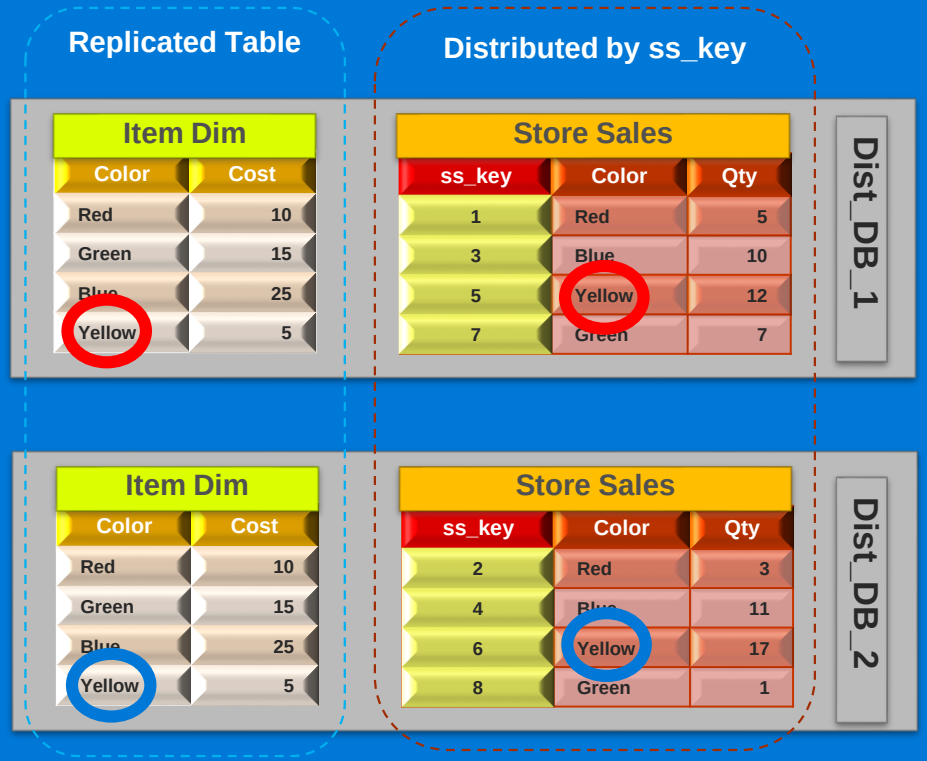
- Join includes compatible distribution keys with compatible data types

Streaming results

- Results streamed to client
- No aggregation (processing) on control node

Distribution Compatible Join with Replicated Table

```
SELECT ss_key, Cost
  FROM item_dim a
  JOIN store_sales b ON a.color = b.color
 WHERE a.color = 'Yellow'
```



Result Set:

5,5

Result Set:

6,5

Final Result Set

5,5 : 6,5

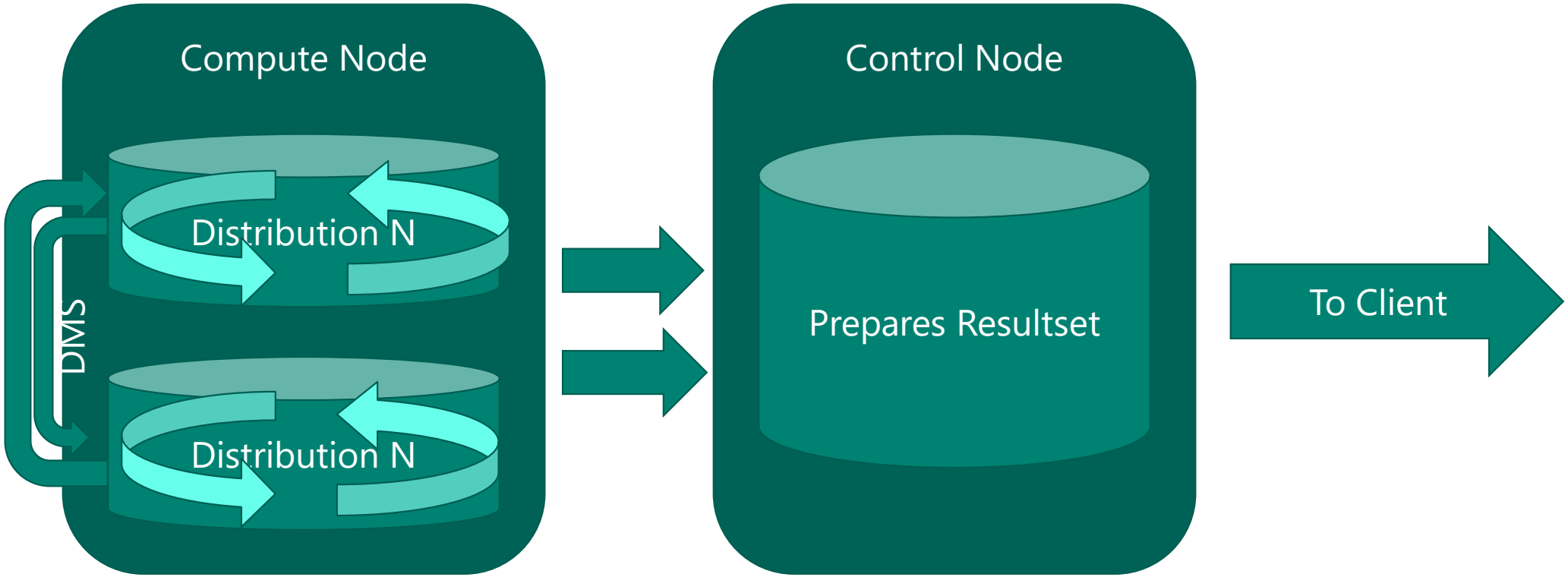
Distribution Compatible Join

- Replication satisfies compatibility for inner joins
- Store Sales distribution key not used

Streaming results

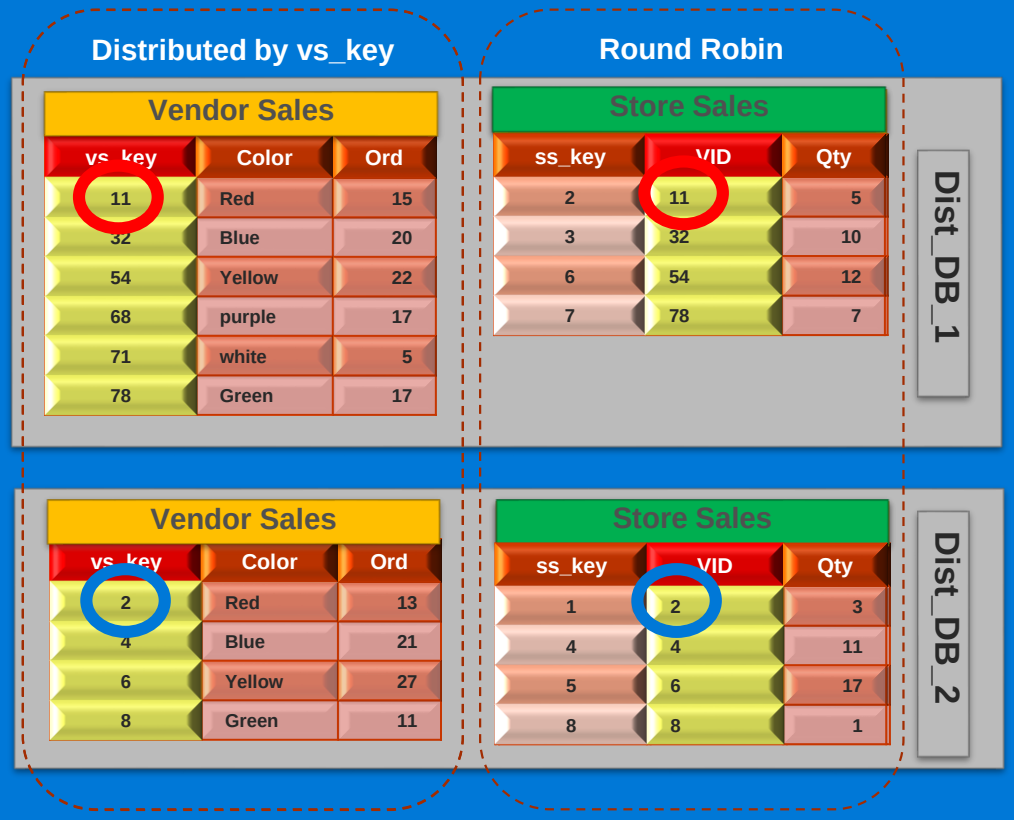
- Results streamed to client
- No aggregation (processing) on control node required

Resultset requires data from different nodes



Distribution Incompatible Join with Round Robin Table

```
SELECT vs_key,  a.ord ,b.qty
  FROM vendor_sales a
  JOIN store_sales b ON a.vs_key = b.VID
 WHERE a.color = 'Red'
```



Result Set
11,15, 5

Result Set
2,13, 3

Final Result Set
11,15,5 : 2,13,3

Distribution Incompatible

- Distribution used from left table (vendor_sales) only

➔ShuffleMoveOperation

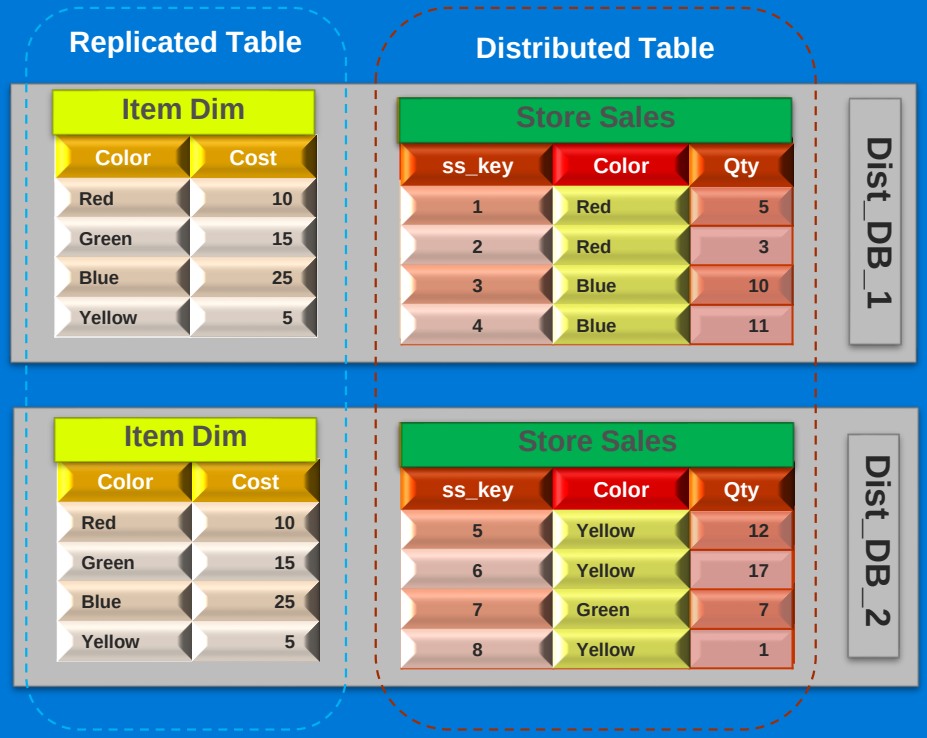
- Data from right table (Store_Sales) is rebuilt with column 'VID' as DK
- Query is now distribution compatible

Streaming results

- Results streamed to client
- No aggregation (processing) on control node

Distribution Incompatible Aggregation

```
SELECT a.color, SUM(a.cost), SUM(b.qty)
  FROM item_dim a
  JOIN store_sales b ON a.color = b.color
GROUP BY a.color
```



Result Set
Red,20,8
Blue,50,21

Result Set
Yellow,15,30
Green,15,1

Final Result Set
Red,20,8 : Blue,50,21 : Yellow,15,30 : Green,15,1

Distribution Compatible Join

- Replication satisfies compatibility for inner joins
- Store Sales distribution key not used

Aggregation: incompatible

- Operation ShuffleMove
- Final results cannot be completed on the compute nodes alone
- Right table (Store Sales) is rebuilt with distribution key='Color' first, then partial aggregation on compute nodes

Streaming results

- No aggregation (processing) on control node
- Results streamed to client

Recommendations

- Round robin only for small tables and hash for large tables
  - Try to align fact and dimension data across distributions
- Round robin tables always require data movement in the system
- Hash tables run in parallel

Data Movement Service

- Data movement round robin vs hash
- Demo

Example: Most Optimal plan

No aggregations

Distribution compatible join

Just a return operation "pass-through"

```
<?xml version="1.0" encoding="utf-8"?>
<dsql_query number_nodes="4" number_distributions="60" number_distributions_per_node="15">
 <sql>select
   top 10 *
from
    lgrc.orders,
    lgrc.lineitem
where
    l_orderkey  =  o_orderkey
    and o_orderdate BETWEEN '1997-01-01' AND '1997-12-31'
order by
    o_orderdate, l_shipdate</sql>
  <dsql_operations total_cost="0" total_number_operations="1">
    <dsql_operation operation_type="RETURN">
      <location distribution="AllDistributions" />
      <select>...</select>
    </dsql_operation>
  </dsql_operations>
</dsql_query>
```

Query

1 step

Search for dsql_operation operation_type

Optimal Plans tend to have fewer steps

Example: Less optimal

```
<dsql_operations total_cost="3376410.23822858" total_number_operations="9">
  1 <dsql_operation operation_type="RND_ID">...</dsql_operation>
  2 <dsql_operation operation_type="ON">...</dsql_operation>
  3 <dsql_operation operation_type="SHUFFLE_MOVE">
      <operation_cost cost="32[illegible]5932.83610284" acc[illegible]83610284"
      <source_statement>...</source_statement>
      [TEMP_ID_287]</destination_ta
      1_orderkey;</shuffle_columns>
  4 ...ation_type="RND_ID">...</dsql_
  5 ...ation_type="ON">...</dsql_oper
  6 <dsql_operation operation_type="SHUFFLE_MOVE">
      <operation_cost cost="100477.402125744" accumulative_cost="3376410.23822858"
      <source_statement>...</source_statement>
      <destination_table>[TEMP_ID_288]</destination_table>
      <shuffle_columns>o_orderkey;</shuffle_columns>
    </dsql_operation>
  7 <dsql_operation operation_type="RETURN">
      <location distribution="AllDistributions" />
      <select>...</select>
    </dsql_operation>
  8 <dsql_operation operation_type="ON">...</dsql_operation>
  9 <dsql_operation operation_type="ON">...</dsql_operation>
</dsql_operations>
```

9 steps

Create/drop table (Always a HEAP table)

Random ID (creates a random number to name temp tables etc)

Important steps

- 2 shuffles
- 1 return

Overhead steps

- Rnd_Id
- Creates/Drops

Same query as last example except now both underlying tables are round robin

Data Movement Types for a Query

DMS Operation	Description
ShuffleMoveOperation	Distribution → Hash algorithm → New distribution Changing the distribution column in preparation for join.
PartitionMoveOperation	Distribution → Control Node Aggregations - count(*) is count on nodes, sum of count
BroadcastMoveOperation	Distribution → Copy to all distributions Changes distributed table to replicated table for join.
TrimMoveOperation	Replicated table → Hash algorithm → Distribution When a replicated table needs to become distributed. Needed for outer joins.
MoveOperation	Control Node → Copy to all distributions Data moved from Control Node back to Compute Nodes resulting in a replicated table for further processing.
RoundRobinMoveOperation	Source → Round robin algorithm → Distribution Redistributes data to Round Robin Table.

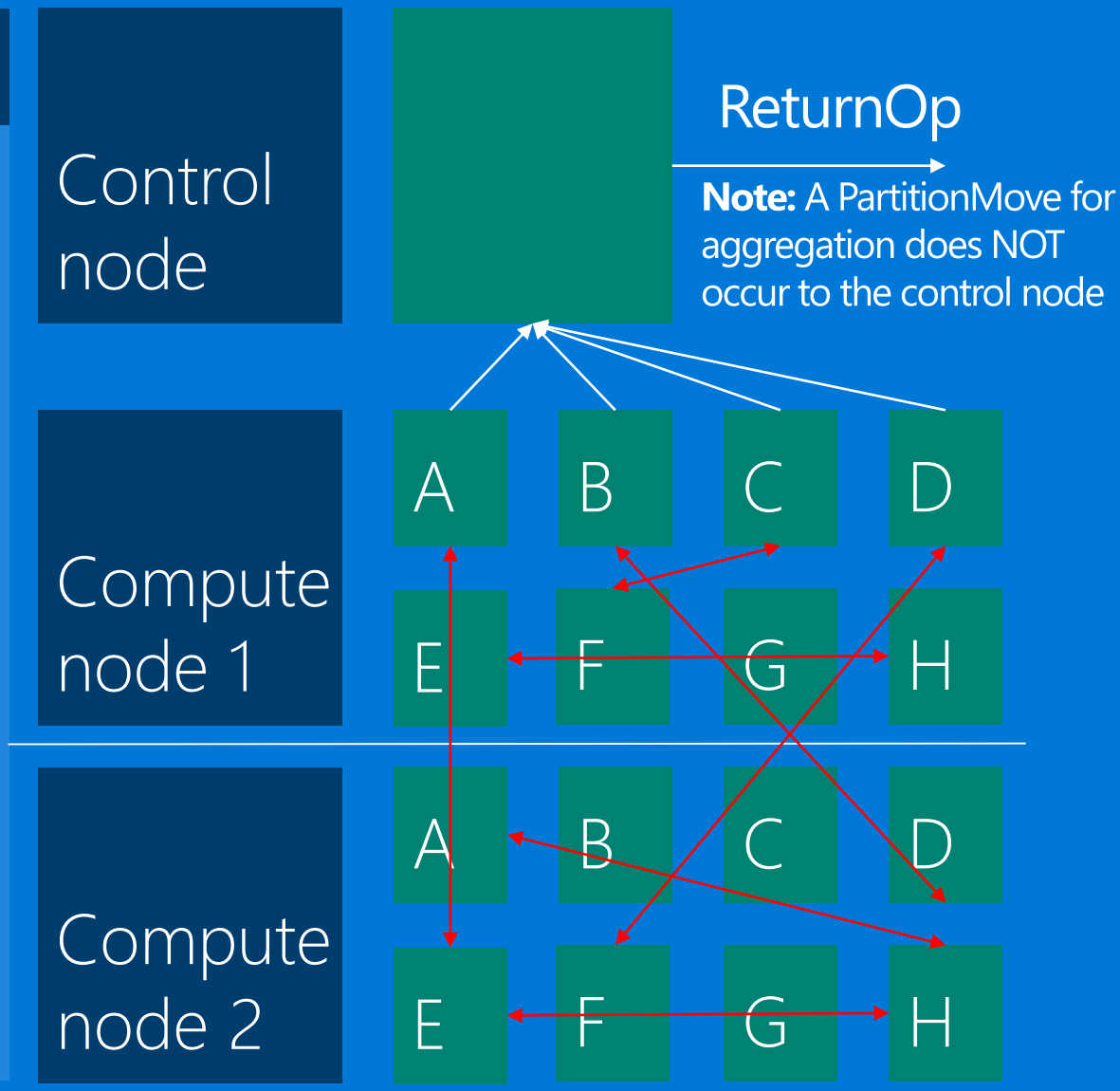
Data Movement Types Demo

Data Movement

Shuffle Move

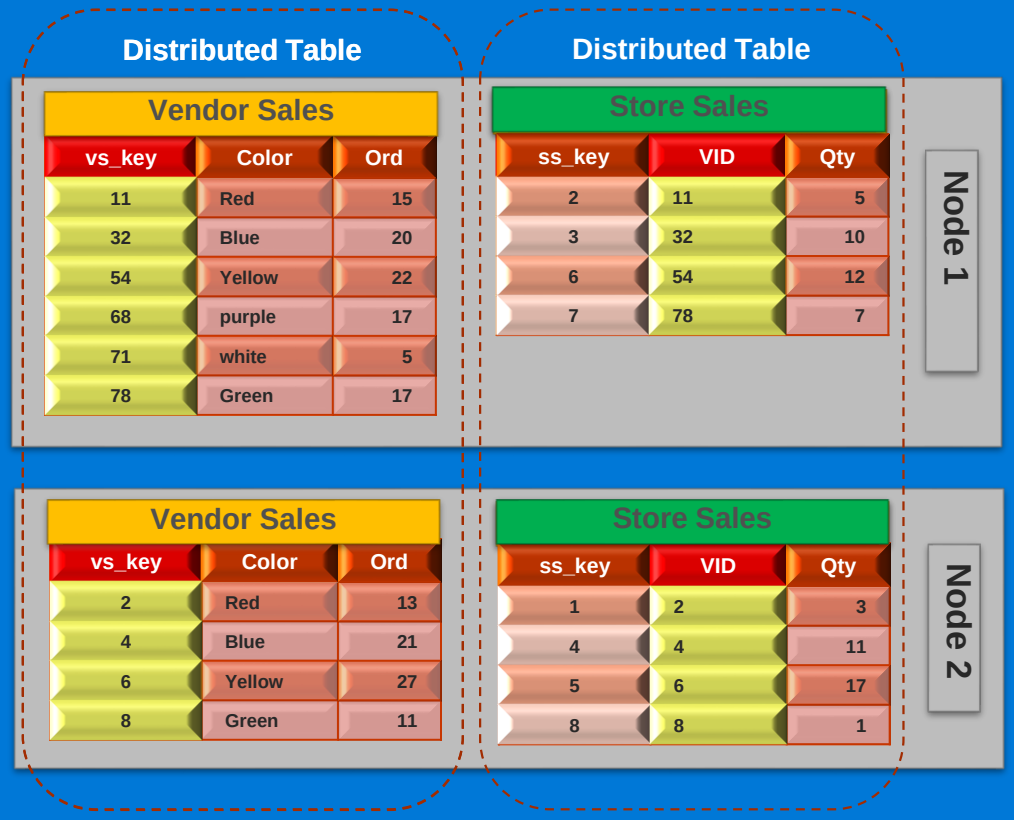
The Shuffle Move is a very common operation and is used to redistribute data between **compute nodes** into a HASH distributed table, in most cases.

- Often the MPP DWH Engine will need to redistribute data to further aggregate it or to perform a join between tables.
- The Shuffle Move will create a new **HASH** distributed table that is hashed on a different column to the original table.
- The column to distribute on for the new table will be determined by the MPP DWH engine based on the query.
- Consider the following query, where the table is distributed on "OrderDateKey" but the query aggregated on "ProductKey":
  - SELECT COUNT(*) FROM FactInternetSales GROUP BY ProductKey



Data Movement: Shuffle

```
SELECT vs_key,  a.ord ,b.qty
  FROM vendor_sales a
  JOIN store_sales b ON a.vs_key = b.VID
 WHERE a.color = 'Red'
```



Result Set
11,15, 5

Result Set
2,13, 3

Final Result Set
11,15,5 : 2,13,3

Data move type: redistribution

- Tables are not co-located on their respective distribution keys

Distribution: incompatible

- Distribution used from left table (vendor_sales) only

ShuffleMove cheaper than BroadcastMove

➔ShuffleMoveOperation

- Data from right table (Store_Sales) is rebuilt with column 'VID' as DK
- Query is now distribution compatible

Streaming results

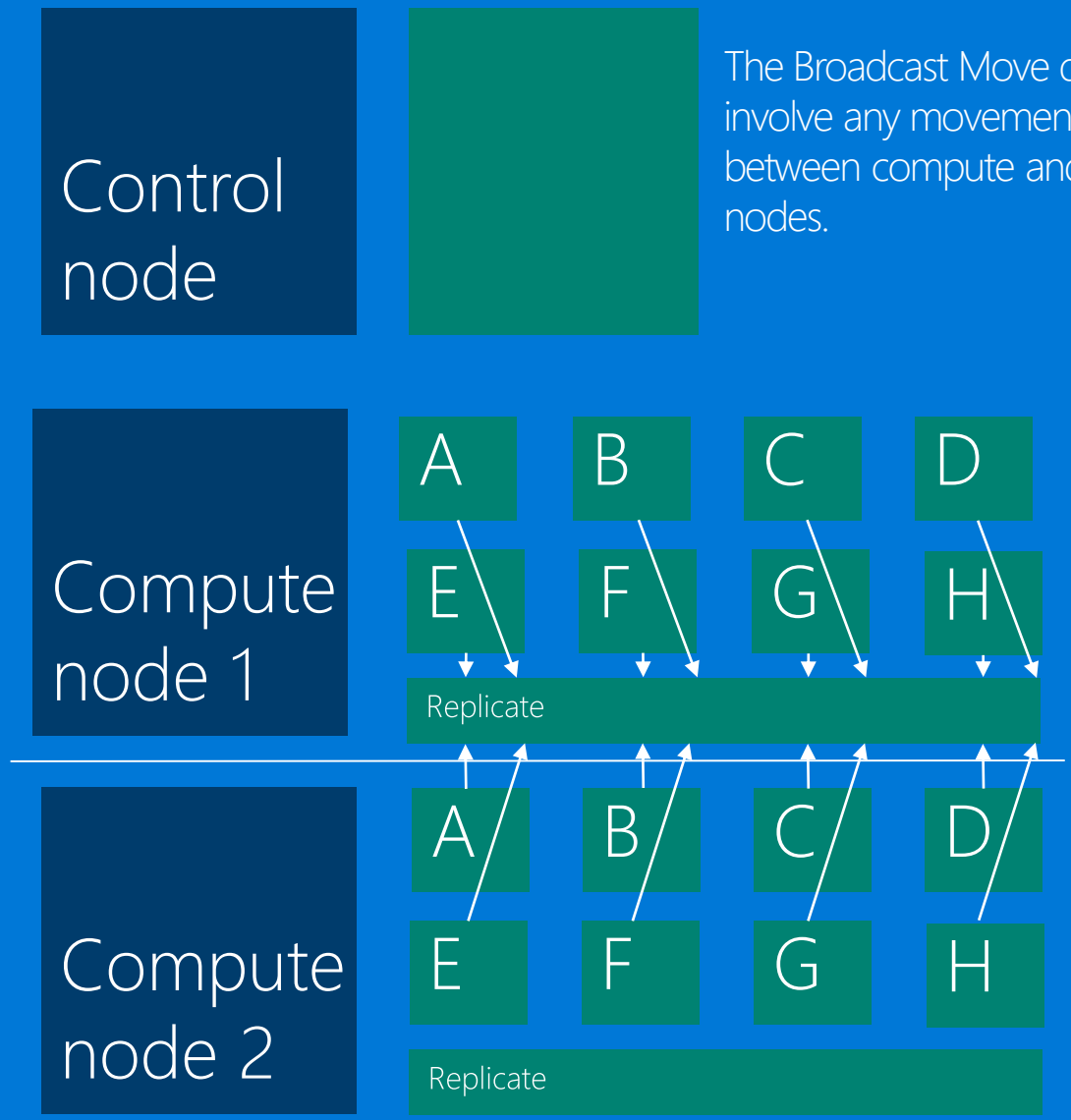
- Results streamed to client
- No aggregation (processing) on control node

Data Movement

BroadcastMove

Scenario 1: The BroadcastMove may be used to move **a few records** or a **few columns** from a HASH distributed table for any further processing...

- Typically the Broadcast Move occurs during query execution when the MPP Optimizer determines it is more efficient then any other DMS operation.
- It may also occur during the load phase if the process is designed to derive replicated tables from a distributed table.

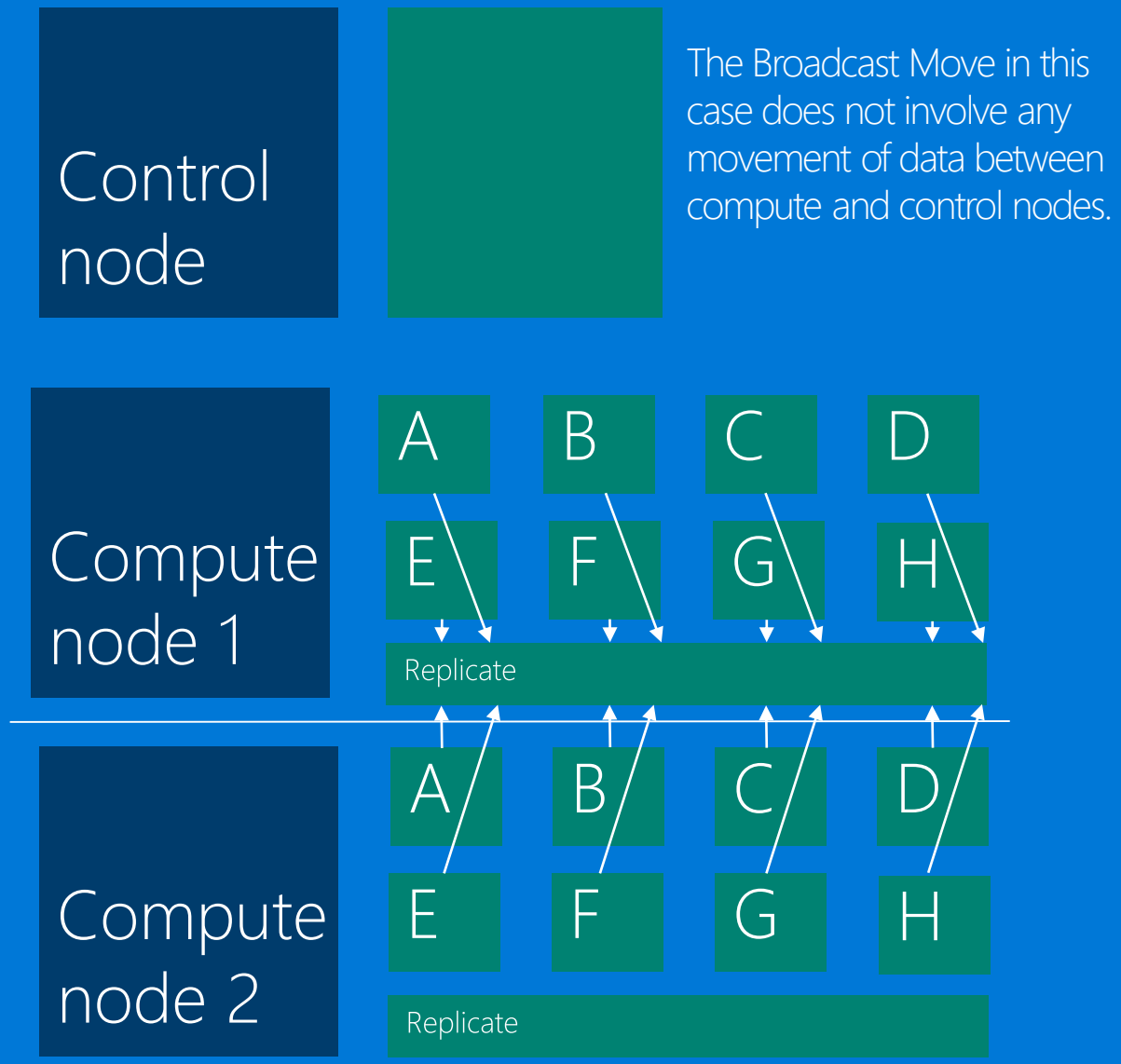


Data Movement

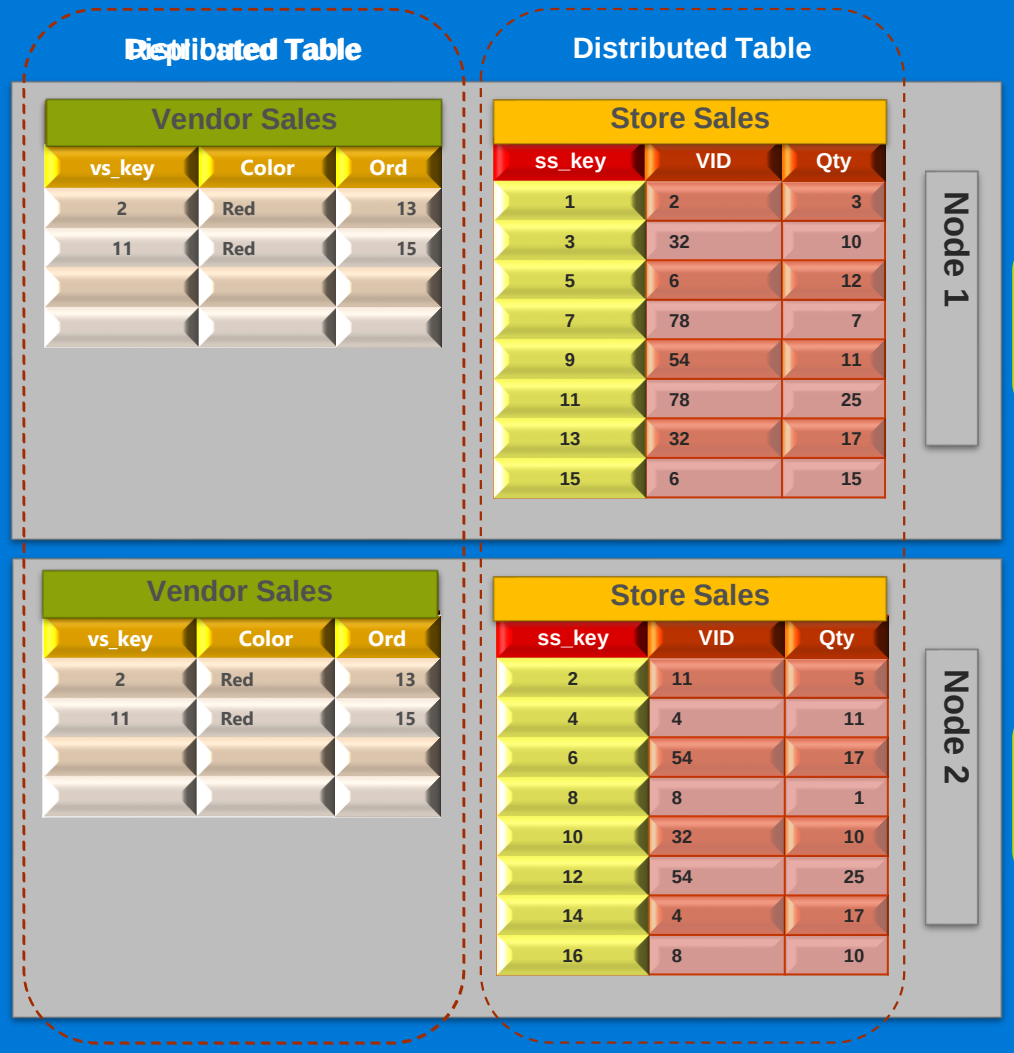
BroadcastMove

Scenario 2: The BroadcastMove is used to move records from a HASH distributed table to create a replicated table. (APS only)

- Typically in this case, the MPP DWH optimizer always decides to use a BroadcastMove.



Data Movement: Broadcast



Result Set
2,13, 3

Result Set
11,15, 5

Final Result Set
2,17,3 : 11,15,5

```
SELECT vs_key,  a.ord ,b.qty
  FROM vendor_sales a
  JOIN store_sales b ON a.vs_key = b.VID
 WHERE a.color = 'Red'
```

Operation type: broadcast

- Tables are not co-located on their respective distribution keys
- Cost for broadcast is lower

Distribution: incompatible

BroadcastMove cheaper than ShuffleMove

➔Broadcast operation

- Distributed Vendor Sales table is getting replicated
  - WHERE predicates pushed down
- Query is now distribution compatible
  - Distributed to replicated

Streaming results

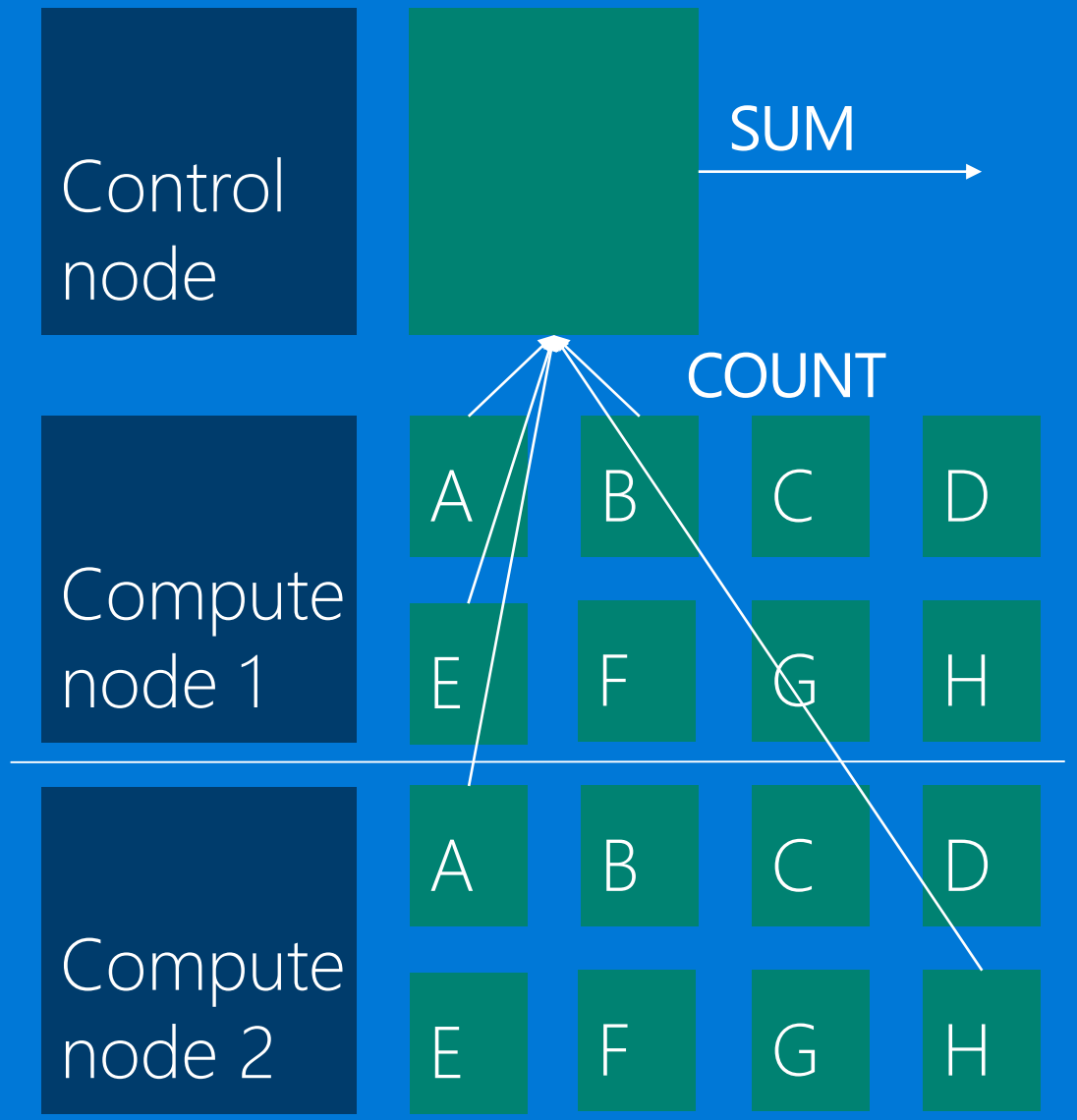
- Results streamed to client
- No aggregation (processing) on control node

Data Movement

PartitionMove

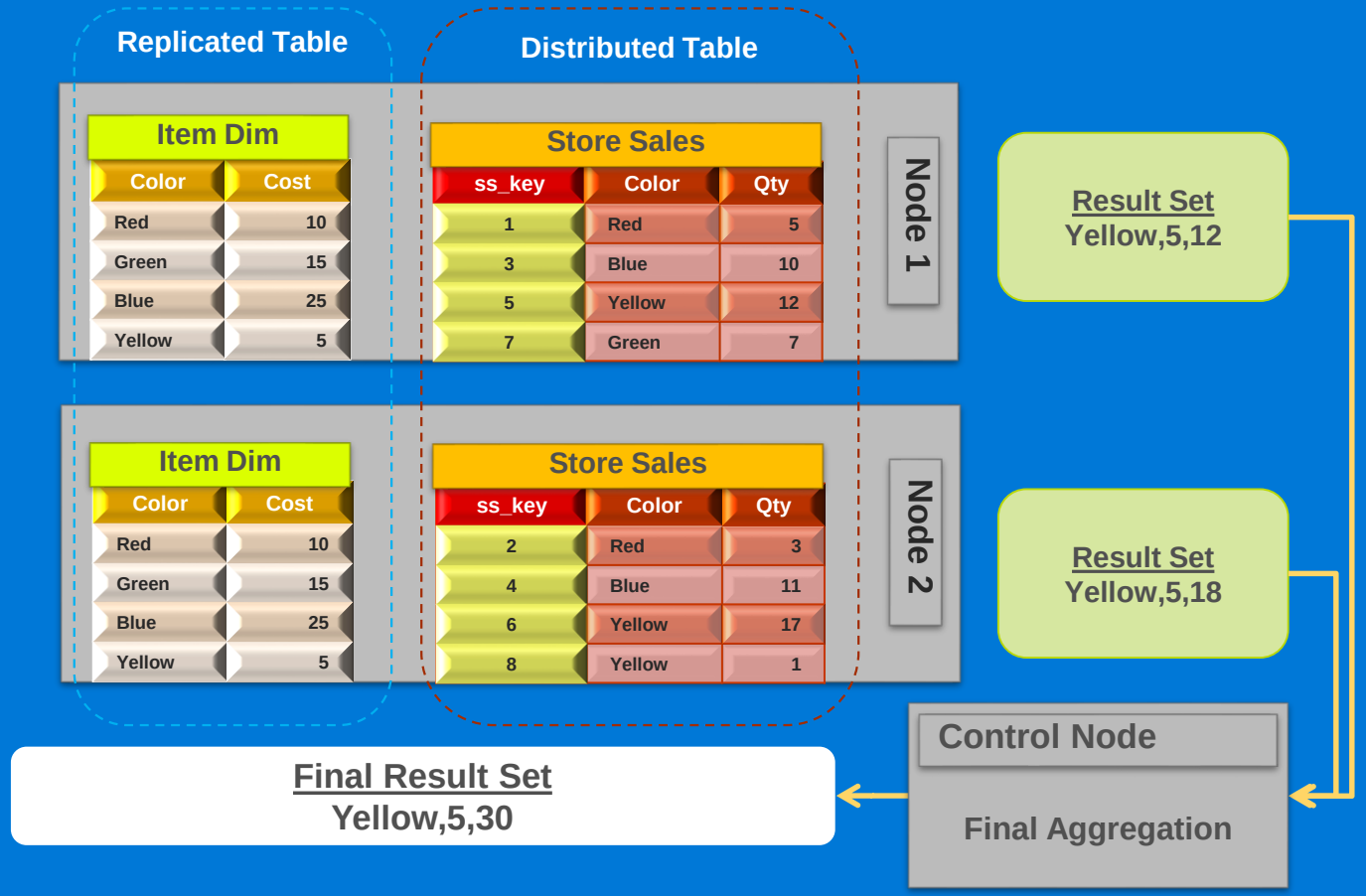
The Partition Move is used to move records from each **compute node** up to the **control node**.

- Consider the following query as an example:
  - SELECT COUNT(*) FROM DistributedTable
- The MPP DWH Engine will rewrite this to send a specific query to each distribution.
- Each distribution will then return the count of rows in that distribution as a single value.
- The SUM of each distribution row count will represent the total row count of the distributed table.
- The MPP DWH Engine needs to perform this SUM operation in a single location, so a **Partition Move** is used to store the intermediate results on the control node database for final aggregation.



Partition Move

```
SELECT a.color, AVG(a.cost), SUM(b.qty)
  FROM item_dim a
  JOIN store_sales b ON a.color = b.color
 WHERE a.color = 'Yellow'
 GROUP BY a.color
```



Join Type: shared nothing
Distribution: compatible

- Replication satisfies compatibility for inner joins
- Store Sales distribution key not used

Aggregation: incompatible

PartitionMove cheaper than ShuffleMove

➔PartitionMove to control node

- Partial aggregations performed on compute nodes and partition-moved to control node

Streaming results

- Results: finally aggregated on the control node
- Results streamed to client

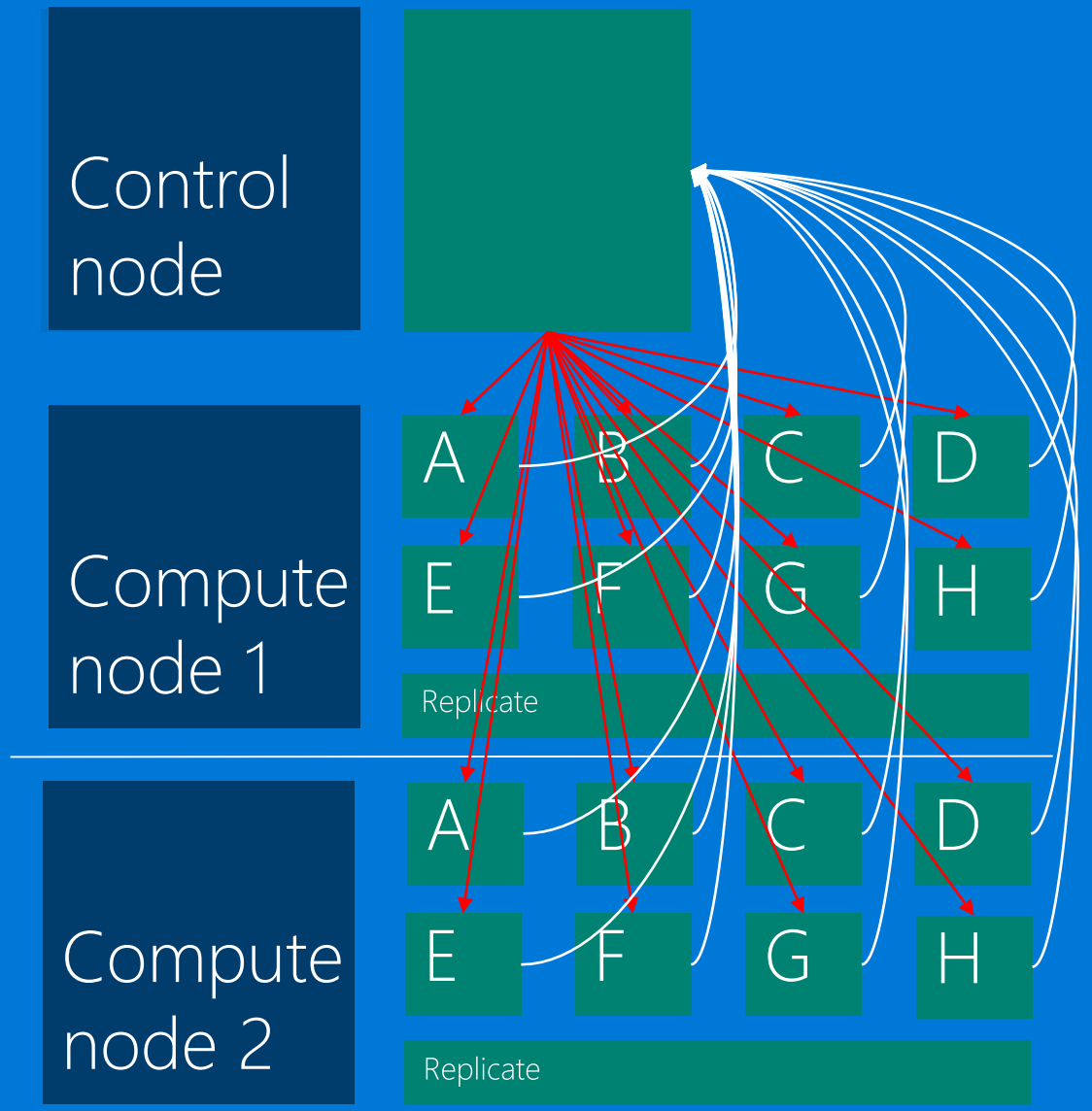
Data Movement

TrimMove

Scenario 1: The TrimMove is used to create a HASH distributed table from a ROUND_ROBIN distributed table, in some cases.

Consider the location of the source and target rows for this movement type.

- The Trim Move processes with hash all values in the source table column that has been defined as the target table HASH column.



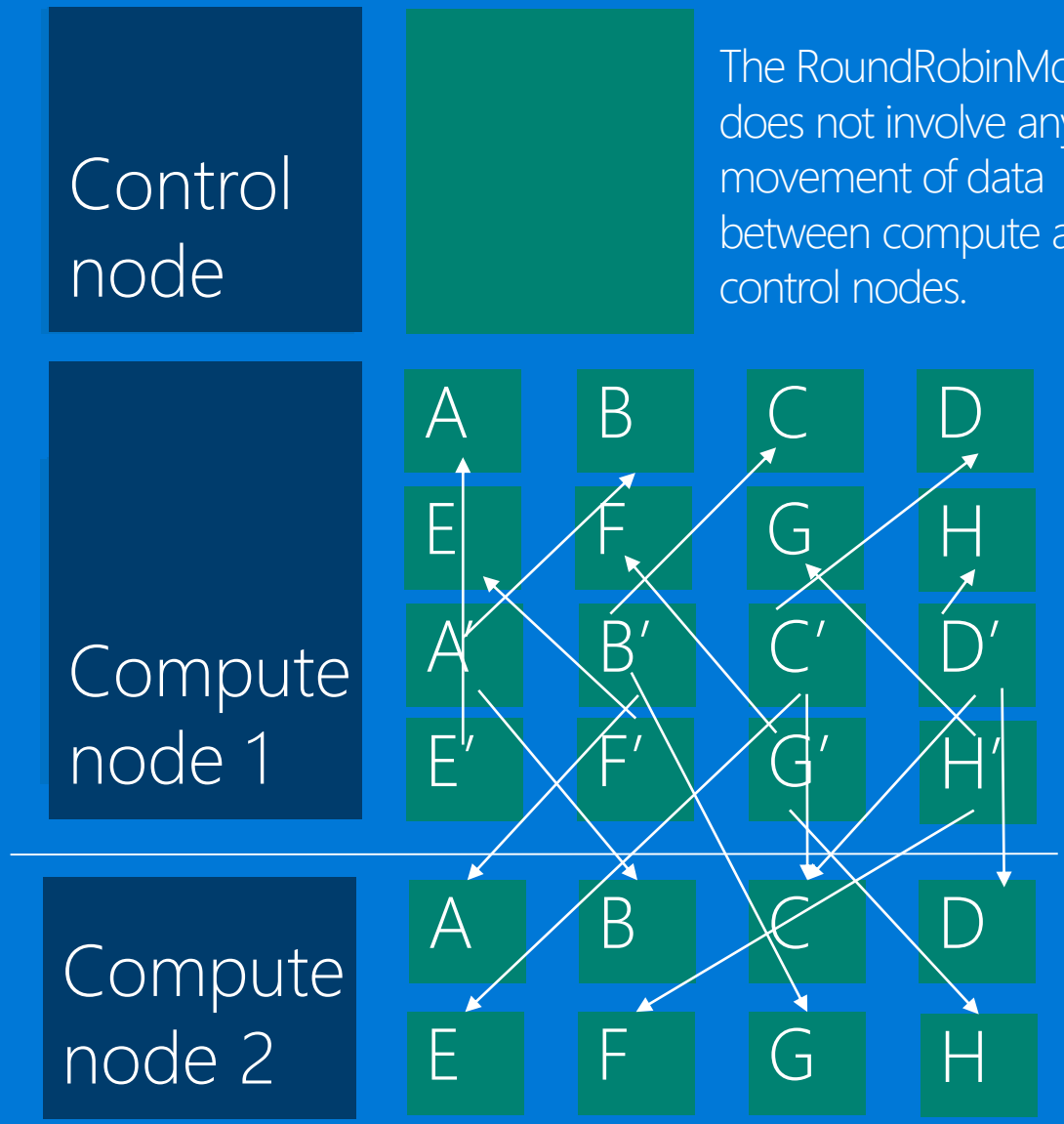
Data Movement

RoundRobinMove

The RoundRobinMove is used to move **records** from **distributed table** into a created **ROUND_ROBIN** distributed table, in most cases.

Consider the location of the source and target rows for this movement type.

- The RoundRobinMove processes uses a round robin algorithm to distribute data. No HASH function will be used against any table column.



Questions to ask when looking at DSQL plan

- What is taking the longest?
  - What is this step doing?
  - Does it make sense? This takes some time to learn.
- What objects are being used by the query?
  - Tables
    - Statistics?
    - Distribution? Hash or Round Robin?
    - How many rows? Skew? DBCC PDW_SHOWSPACEUSED
  - External Tables
    - No push down, consider loading
  - Views
    - What's under the view?
    - Overloaded with joins?

Key best practices

- Create and update statistics
  - Statistics must be manually created and maintained.
  - Create statistics on all columns in join, group by and where
  - Add multi-column statistics where join on multiple columns or predicates
- Hash distribute large tables
  - Selecting the right distribution columns will minimize data movement
- Use resources classes thoughtfully
  - Balance need for memory with need for concurrency
  - Not all queries benefit
- Load large external tables rather than query
  - All data is brought back, no push down

Performance Investigation Guide

- Look at the system
  - A lot of users/high concurrency?
  - Are resource classes being correctly used
  - Are queries faster with a single user?
  - Loading happening at the same time as queries?
- Look at query plans
  - Find longest running steps
  - Is the DMS being invoked

Maintenance

- Use Azure Automation to run jobs
- Use a large or extra large role for maintenance
- Use partitions
  - But not too many
- Rebuild Indexes
  - Think about loading strategy
- Create statistics
- Rebuild statistics
- If you are pausing do not pause and resume every hour

Major factors that cause performance issues

- Statistics
- Data Skew
- Clustered Columnstore Index Maintenance (lack of)
- Poor design choices
  - Distributed vs. Replicated vs. Round Robin tables
  - CCI vs. RowStore vs Heap
  - Query Design
- Avoid functions that may cause the system to evaluate the data row by row


Microsoft