Table Design

Agenda

- Table Creation Process
- Table Structure
- Table Geometry
- Data Definition Language
- Temporary Tables
- Virtual Tables
- Using CTAS
- Table Design

Table Creation Process

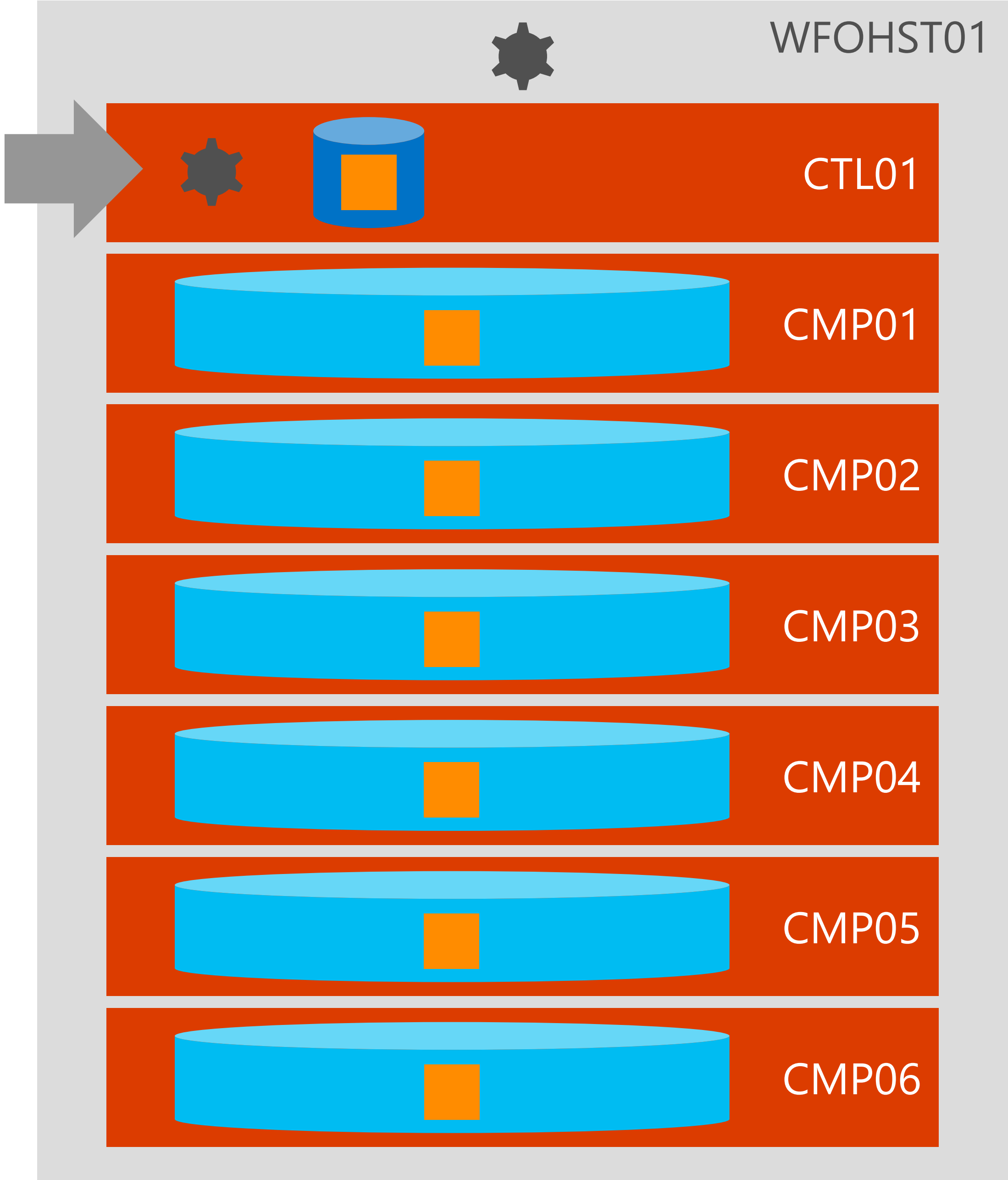
Creating a Table

User Submits Create Table

```sql
CREATE TABLE Dimension
( Dim_ID INT NOT NULL
, Dim_DK INT NOT NULL
, Dim_name char(20) NOT NULL
)
WITH
( CLUSTERED INDEX (Dim_ID)
, DISTRIBUTION = REPLICATE
);
```

PDW

- Checks permissions
- Creates table (physical)
- Creates clustered index on table (physical)
- Creates table (logical)
- Adds extended property for index name (logical)
- Creates clustered index (logical)
- Adds extended properties (logical)
- Defaults table level statistics (logical)



Extended Properties

Table Extended Properties	Value
pdw_physical_name	<Name of table>
pdw_distribution_type	<Replicated or Distributed>
pdw_distribution_column	If replicated then empty else name of distribution column

Index Extended Property	Value
pdw_physical_name_index_Idx_<guid>	<Name of index> If system generated ClusteredIndex_<guid> If user named then name given

Process Summary

Table creation is both logical & physical

- Logically (control) in shell database
- Physically (compute) in distributed database
- All table data is held physically
- No user data is persisted logically

Table Structure

Sample DDL

```
CREATE TABLE item
(     i_item_sk         INTEGER    not null
,     i_item_id         CHAR(16)   not null
,     i_rec_start_date DATE
,     i_rec_end_date    DATE
,     i_product_name    CHAR(50)
,     i_item_desc       VARCHAR(200)
,     i_current_price   DECIMAL(7,2)
,     i_wholesale_cost DECIMAL(7,2)
,     i_brand_id        INTEGER
,     i_brand           CHAR(50)
,     i_category_id     INTEGER
)
WITH
(CLUSTERED INDEX (i_item_sk)
,DISTRIBUTION=replicate
);
```

```
CREATE TABLE store_sales
(ss_sold_date_sk          int
,ss_sold_time_sk          int
,ss_item_sk               int    not null
,ss_customer_sk           int
,ss_store_sk              int
,ss_promo_sk              int
,ss_ticket_number         int    not null
,ss_quantity              int
,ss_sales_price           decimal(7, 2)
)
WITH
(     CLUSTERED COLUMNSTORE INDEX
,     DISTRIBUTION=HASH (ss_item_sk)
,     PARTITION (ss_sold_date_sk
                RANGE RIGHT FOR VALUES ()
                )
);
```

Storage Format Options

Row Store
- Heap
- Clustered Index
- Non Clustered Index

Also supports
- Partitioning
- Page Compression

Column Store
- Read/Write
- Also uses a row store
  - Incremental change (deltas)
  - Clustered Index on RowID
- Partitioning
  - Cannot split partitions with data
- Compression
  - Columnar compression

Row Store: Heaps

- Just a collection of pages
- Append only
- No ordering
- Minimal fragmentation
- Fast target for loading
- Good for partition scan queries (MOLAP SSAS)

Understanding Heaps

Deletes are not physically removed from a heap until it is rebuilt. As data is added SQL Server allocates more pages to the table.

Row Store: Clustered Index

- Implemented using a Balanced Tree (b-tree)
- Physically orders data in the table
- Leaf level of the index is the data
- Get fragmented over time
- Good for limited range scan queries

Row Store: Non-Clustered Index

- Implemented using a Balanced Tree (b-tree)
- Physically orders data in the index
- Leaf level of the index points to the data

Row Store: Non-Clustered Index

Pros

- Good for singleton selects
- Helpful in some ROLAP scenarios

Cons

- Easily fragmented
- Can generate lots of random I/O
- Impacts sequential scans
- Impact write throughput
- Secondary index adds to storage need

Heaps vs. Clustered Index

Use Heaps For

- Fast Loads
- Partition scan queries
- Minimal Fragmentation

Use Ordered Data For

- Sequential data layout
- Selective queries
- Range scan queries

Column Store: Taxonomy

Not really a clustered index – misnomer

- A Column Store is really just a different storage format for data
- Replaces the clustered index b-tree hence the name
- Syntax is CLUSTERED COLUMNSTORE INDEX

Column Store: Taxonomy

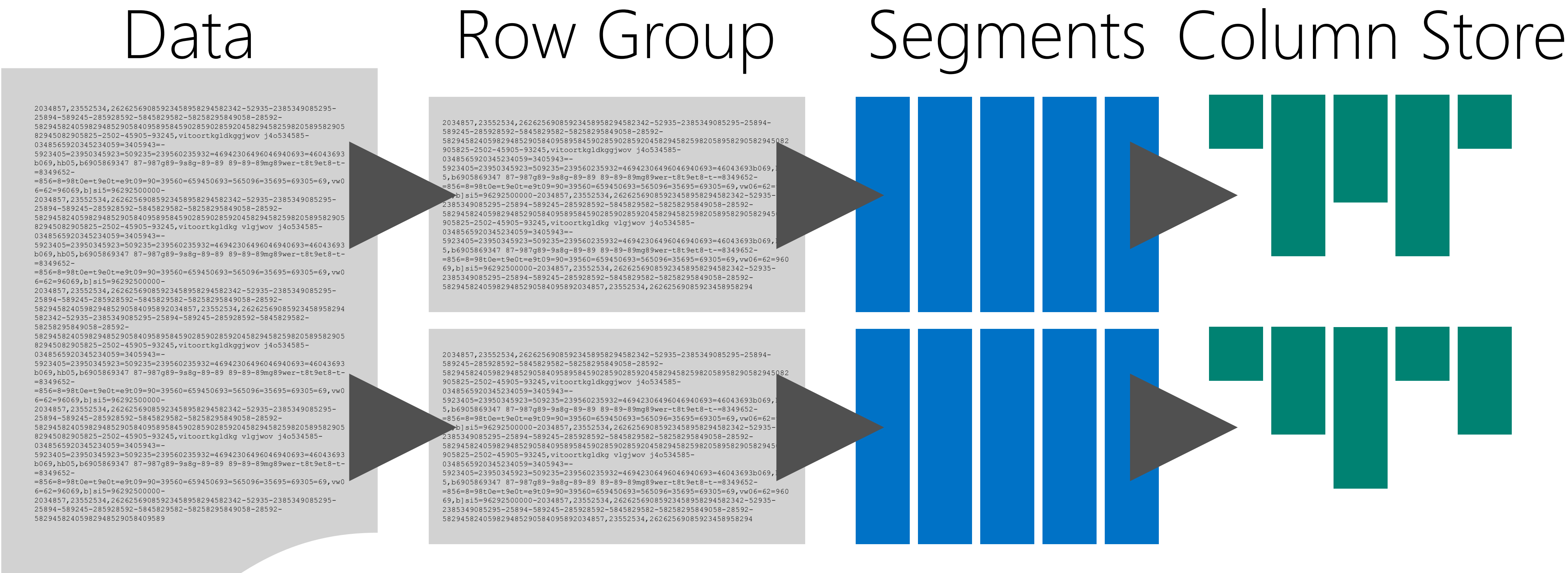
Rows first grouped into batches of ~1M rows

- This is called a row group

Rows then re-ordered for optimal compression

- Columns compressed individually up to 15x
- Each column is split into segments

Column Store Taxonomy Explained



sys.pdw_nodes_column_store_row_groups

sys.pdw_nodes_column_store_segments

sys.pdw_nodes_column_store_dictionaries

Column store: Taxonomy Continued

Column stores consist of two storage “areas”

- Column Store itself
- Delta Store

Column store: Taxonomy Continued

Delta store is a page compressed b-tree

- It is a row store
- Small volume (# rows) changes will see rows first enter the delta store
- Batches >= 102,400 per affected partition will see rows written directly to the column store as a new row group

Small DML against the column store

- Inserts below the threshold write into the delta store
- Deletes are marked logically in the column store
- Deletes are processed physically against data in the delta store
- Deleted rows are only purged during an Index Rebuild
- Updates are converted to a logical delete and an insert

Delta Store

Delta stores are

- Affiliated to one table
- Partition aligned
- Page Compressed Row groups
- Only created when needed for data changes

Delta Store

Facts & Figures

- Zero, one or more delta stores per partition
- Maximum of 1,048,576 rows per delta store
- 1,048,576 = max # of rows in a row group

Row Groups

Exist in one of three states

- Open: In delta store; accepting new rows
- Closed: In delta store; not accepting new rows
- Compressed: In Column Store format

Delta Store to Column Store

Once a row group has closed it can be converted to column store format:

As an online operation

- Tuple Mover: a background process
- ALTER INDEX..REORGANIZE

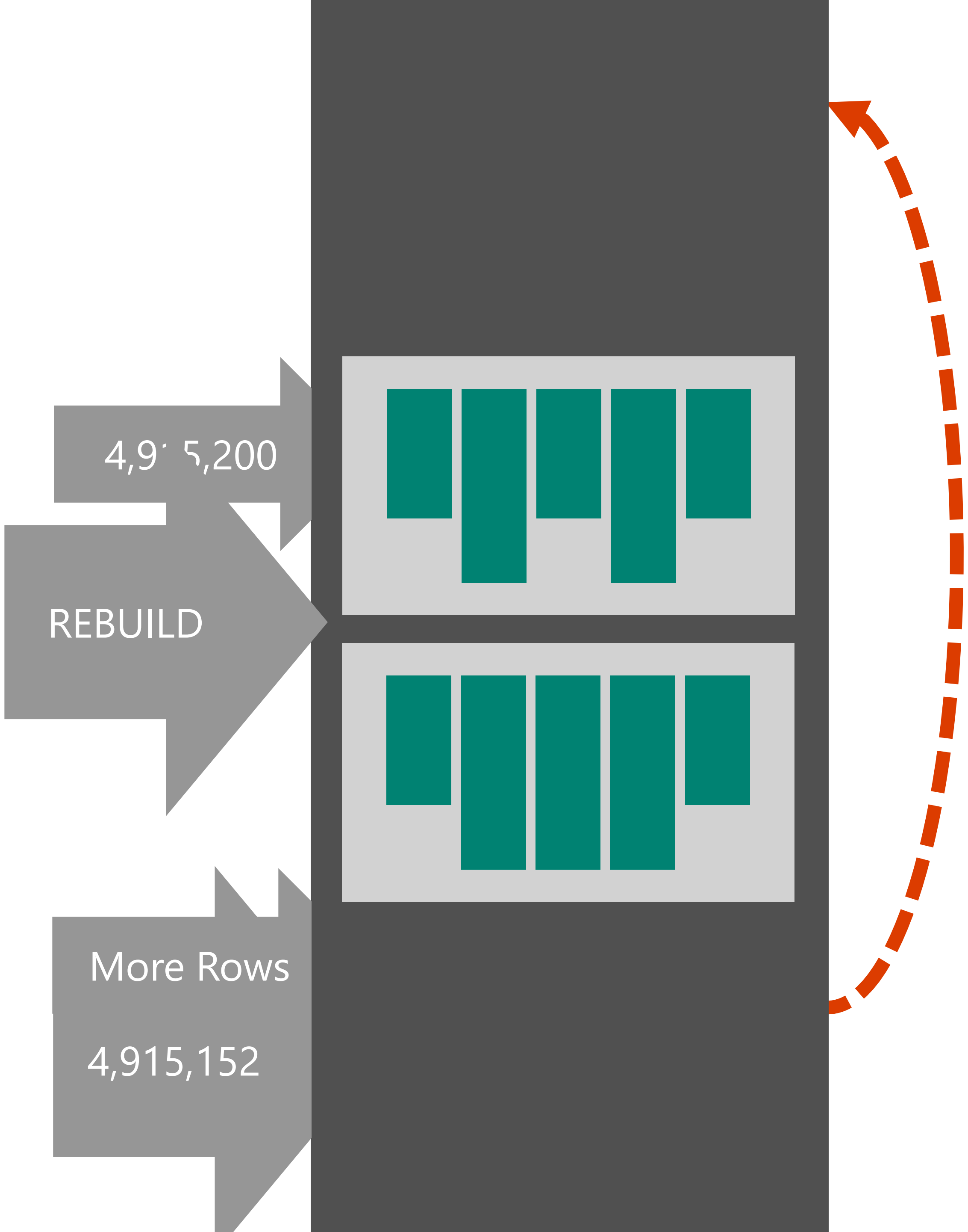
As an offline operation

- ALTER INDEX..REBUILD

Delta Store to Column Store

REBUILD	REBUILD	REORGANIZE
Re-compresses the entire partition or table	Therefore also affects open row groups	Has absolutely no affect on open row groups

Index Rebuild



Assumptions
- 6 node appliance
- Table is distributed
- Data evenly spread
- Table is partitioned

Min batch size to write to columnstore = 4,915,200

- 4,915,152 rows written in a single batch
- Rows enter delta stores
- Delta Store closed
- Tuple Mover moves closed row group and converts it to column store
- Delta Store is removed

- After Index Rebuild initiated
- Table taken offline
- Data recompressed
- All row groups now in columnstore

Column Store Benefits

Enhanced Compression

- Page compression ratio up to 3.5:1
- Column store compression ratio up to 15:1

What does this mean?

- Greater ROI for Available Storage
- Greater throughput from reduced physical I/O
- Isolate columns to reduce I/O further

Column Store Metadata

```
SELECT
 t.name          AS logical_table_name ,rg.partition_number
,t.[object_id]   AS logical_object_id  ,rg.row_group_id
,i.[name]        AS logical_CCI_Name   ,rg.delta_store_hobt_id
,nt.pdw_node_id  AS pdw_node_id        ,rg.[state]
,nt.[name]       AS physical_name      ,rg.state_description
,nt.[object_id]  AS physical_object_id ,rg.total_rows
,ni.[name]       AS physical_CCI_Name  ,rg.size_in_bytes
FROM sys.tables t
JOIN sys.indexes i                                ON  t.object_id      = i.object_id
JOIN sys.pdw_index_mappings im                    ON  i.object_id      = im.object_id
                                                  AND i.index_id       = im.index_id
JOIN sys.pdw_nodes_indexes ni                     ON  im.physical_name = ni.name
                                                  AND im.index_id      = ni.index_id
JOIN sys.pdw_nodes_tables nt                      ON  nt.object_id     = ni.object_id
                                                  AND nt.pdw_node_id   = ni.pdw_node_id
JOIN sys.pdw_nodes_column_store_row_groups rg ON  ni.object_id     = rg.object_id
                                                  AND ni.index_id      = rg.index_id
                                                  AND ni.pdw_node_id   = rg.pdw_node_id
```

Column Store Limitations

- Operations on strings are slow
- The predicate is not pushed down

Segment must first

- be brought in from disk
- uncompressed

Before predicate can be applied

Use integer based types for search predicates or consider row storage

Batch Mode Processing

Columnstore Indexes enable “Batch Mode” in the query processor

- Takes advantage of the compressed columnar format
- Operate on 1000 rows at a time
- Dramatically reduces CPU cost in query execution
- Requires a parallel plan and does not support all table operators (e.g. Merge/Nested loop)

Table Geometry

PDW Table Geometry

- Replicated Tables
- Distributed Tables

Replicated Tables

- Cloned on all compute nodes
- One clone is created per compute node
- Can be partitioned
- Replicated table data is available on every compute node

Distributed Table

- One table is divided into many sub-tables
- Each compute node contains 8 sub-tables
- Sub-table can also be partitioned
- Data is then hashed across the appliance into sub-tables
- Only a subset of distributed table data is available on each node

Replicated vs. Distributed

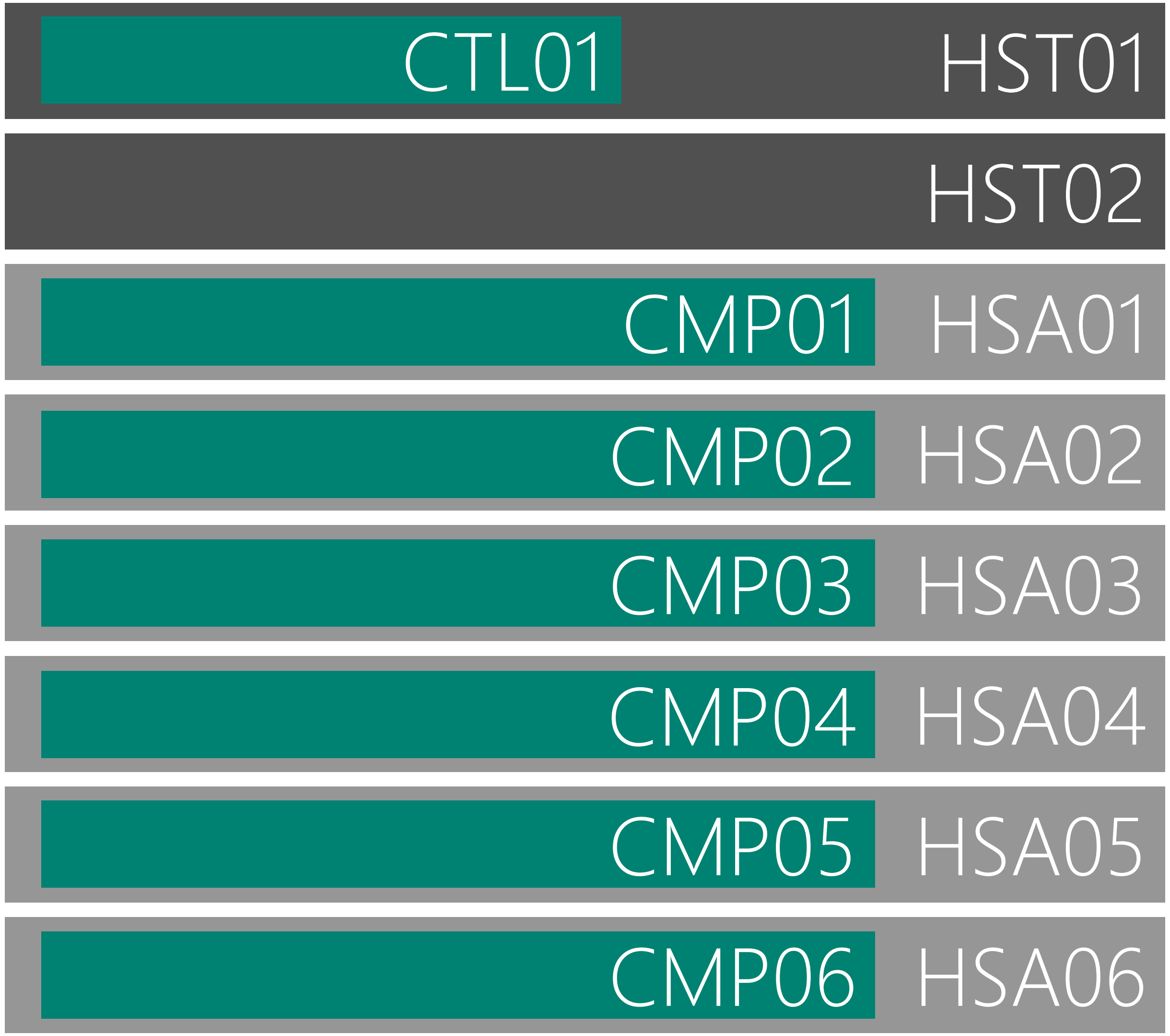


Table Geometry Compared

Replicated

- Full table stored on each compute node
- Table locked on write
- Slow for loading
- JOIN is fast read
- Default table type in CREATE TABLE DDL

Distributed

- Sub-tables data held on each compute node
- Row locked on write
- Fast for loading
- JOIN may need data movement
- Requires HASH distribution option in CREATE TABLE

Examples of Distributed Tables

- Transaction fact tables
- Large Snapshot Tables
- Very Large Dimensions
- High write frequency tables (logging table)

Examples of Replicated Tables

- Dimensions
- Small fact tables
- Static data tables

Geometry vs. Locks

Distributed tables

- Issue exclusive row locks during a write

Replicated tables

- Issue exclusive **table** locks during a write

Writes to replicated tables

- Need to result in completely consistent data in all nodes
- Requires distributed transaction marshalling for rollback
- Significant overhead in replicated table writes – much slower

Data Definition Language

Sample DDL - Distributed

```
CREATE TABLE [dbo].[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 DEFAULT 0
)
WITH ( CLUSTERED COLUMNSTORE INDEX
     , DISTRIBUTION = HASH([FinanceKey])
     );
```

Constant

Geometry

Sample DDL - Replicated

```
CREATE TABLE [dbo].[DimCalendarDate]
(
        [DateKey]                 int       NOT NULL
,       [FullDateAlternateKey]   date      NOT NULL
,       [DayNumberOfWeek]        tinyint   NOT NULL
,       [EnglishDayNameOfWeek]   nvarchar(10) COLLATE Latin1_General_100_CI_AS_KS_WS NOT NULL
,       [DayNumberOfMonth]       tinyint   NOT NULL
,       [DayNumberOfYear]        smallint  NOT NULL
,       [WeekNumberOfYear]       tinyint   NOT NULL
,       [EnglishMonthName]       nvarchar(10) COLLATE Latin1_General_100_CI_AS_KS_WS NOT NULL
,       [MonthNumberOfYear]      tinyint   NOT NULL
,       [CalendarQuarter]        tinyint   NOT NULL
,       [CalendarYear]           smallint  NOT NULL
,       [CalendarSemester]       tinyint   NOT NULL
)
WITH (CLUSTERED INDEX ( [DateKey] ASC )
        ,DISTRIBUTION = REPLICATE
        )
;
```

Index not named

Windows Collation

Create DDL Statements

- Create Table
- Create Table As Select (CTAS)
- Create Remote Table As Select (CRTAS)
- Create External Table
- Create External Table As Select (CETAS)

Alter & Drop Statements

- Alter Table
- Drop Table
- Drop External Table

Table DDL in PDW

- Column name
- Data Type*
- Column Collation
- Null-ability
- Default Constraint*
- Indexing*
- Partitioning
- Temporary Tables*
- Table geometry**

*   partial support
** PDW exclusive

Table DDL not used by PDW

- User-defined Schemas
- Primary keys
- Foreign keys
- Check constraints
- Unique constraints
- Unique index
- Computed columns
- Identity / sequences
- FileTable
- FileStream
- Column Sets
- Sparse Columns
- ROWGUIDCOL
- User Defined Data Types

Automated Syntax

- Filegroup assignment
- Compression

Default Constraints

- CREATE / ALTER TABLE only
- Must be:
  - a constant
  - a string
- Cannot be:
  - an expression
  - a function
  - on distribution key
  - created by CTAS

This Works
This Does Not Work

CREATE TABLE Sales

Why?

How does SQL Server generate constant default constraints?

Parentheses convert a constant to an expression

As expressions!

Column Data Types

Used by PDW

- All integer types
- All character types
- All date types
- All money types
- All binary types
- All floating types
- Bit

Not in PDW

- HierarchyID
- Timestamp
- Uniqueidentifier
- Sql_variant
- Spatial (geography, geometry)
- Synonyms (Numeric, sysname)
- Blobs(text, ntext, image, (MAX))
- User Defined Types

Partitioning

Understanding Partitioning

Partitioning can be applied to

- Replicated Tables
- Distributed Tables
- Heaps
- Clustered Index
- Column Stores

```sql
CREATE TABLE [dbo].[FactInternetSales]
(
     [ProductKey]            int       NOT NULL
,    [OrderDateKey]          int       NOT NULL
,    [DueDateKey]            int       NOT NULL
,    [ShipDateKey]           int       NOT NULL
,    [CustomerKey]           int       NOT NULL
,    [PromotionKey]          int       NOT NULL
,    [CurrencyKey]           int       NOT NULL
,    [SalesTerritoryKey]     int       NOT NULL
,    [SalesOrderNumber]      nvarchar(20) COLLATE Latin1_General_100_CI_AS_KS_WS NOT NULL
,    [SalesOrderLineNumber]  tinyint   NOT NULL
,    [RevisionNumber]        tinyint   NOT NULL
,    [OrderQuantity]         smallint  NOT NULL
,    [UnitPrice]             money     NOT NULL
,    [SalesAmount]           money     NOT NULL
)
WITH
(CLUSTERED COLUMNSTORE INDEX
,DISTRIBUTION = HASH([OrderDateKey])
,PARTITION  ([OrderDateKey] RANGE RIGHT FOR VALUES
              (20000101,20010101,20020101,20030101,20040101,20050101)
            )
);
```

Creating a Table

User Submits Create Table

DW Engine

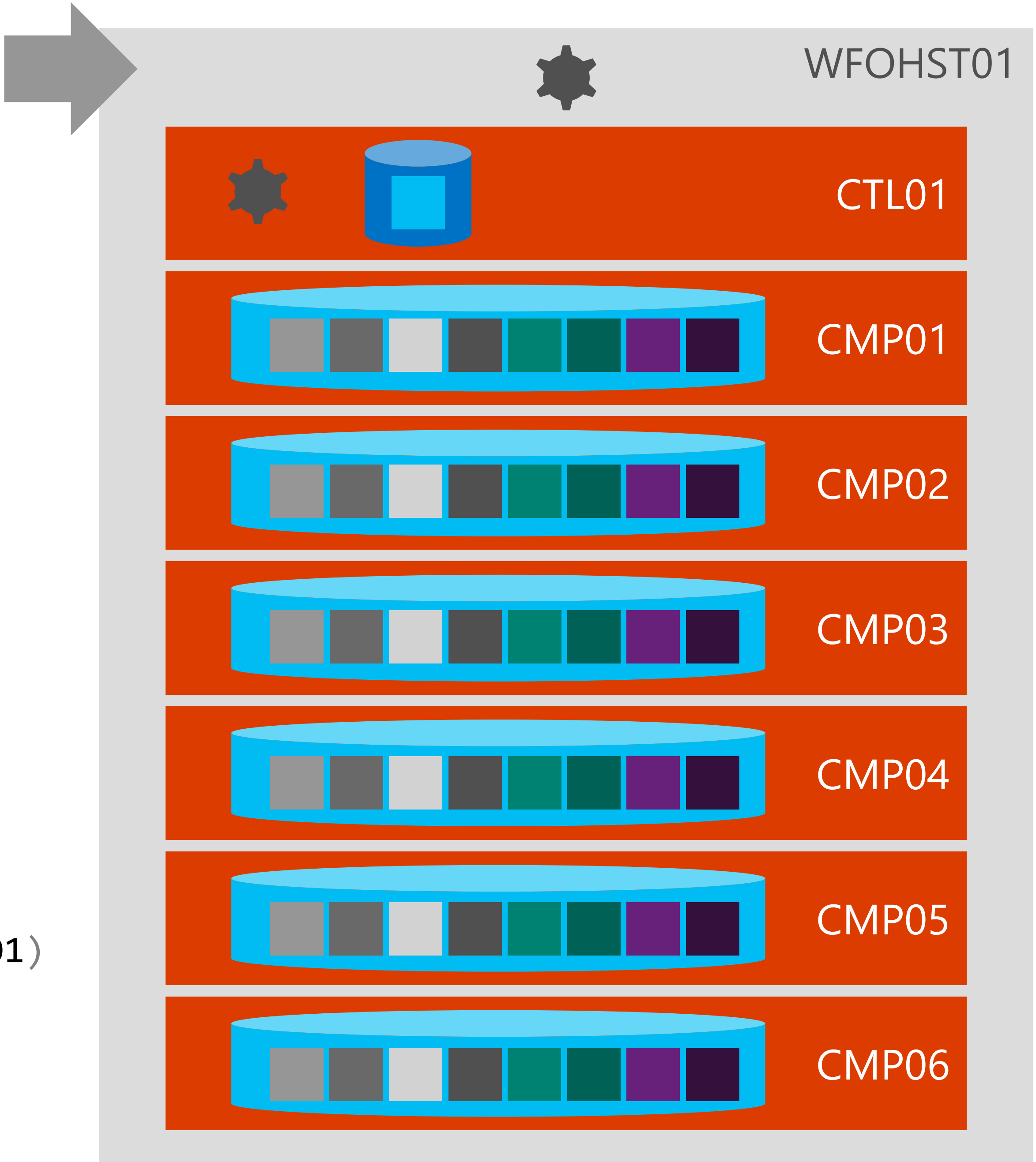
```
CREATE TABLE [dbo].[FactInternetSales]
(
      [ProductKey]              int            NOT NULL
,     [OrderDateKey]            int            NOT NULL
,     [CustomerKey]             int            NOT NULL
,     [PromotionKey]            int            NOT NULL
,     [SalesOrderNumber]        nvarchar(20)   NOT NULL
,     [OrderQuantity]           smallint       NOT NULL
,     [UnitPrice]               money          NOT NULL
,     [SalesAmount]             money          NOT NULL
)
WITH
(CLUSTERED COLUMNSTORE INDEX
,DISTRIBUTION = HASH([OrderDateKey])
,PARTITION   ([OrderDateKey] RANGE RIGHT FOR VALUES
(20000101,20010101,20020101,20030101,20040101,20050101)
             )
);
```

Creates partition scheme (logical)

name (logical)

ded properties (logical)

Defaults table level statistics (logical)



Notes on Partitioning

- All physical tables use same range values
- Range can be right or left

Notes on Partitioning

Distributed tables are “partitioned” by default

- Logical table with 2 partitions actually has 16 physical partitions per compute node
- Important to remember when sizing partitions

Columnstore index divided by partition boundary

- Partition size can affect compression efficiency
- Partition size can impact query performance

Partition DDL

ALTER TABLE

- MERGE
- SPLIT
- SWITCH

Limitations

Table must be created with at least 1 partition

- Cannot be applied after creation

Boundary values represented as literals

- Cannot use a variable
- Can lead to brittle implementations

Column store cannot split / merge partitions containing data

Identifying Partitions

```sql
CREATE VIEW [dbo].[vPartitionMetaData]
AS
SELECT t.name                 AS TableName
,      i.name                 AS IndexName
,      p.partition_number
,      p.partition_id
,      p.rows
,      i.data_space_id
,      f.function_id
,      f.type_desc
,      f.boundary_value_on_right
,      r.boundary_id
,      r.value                AS BoundaryValue
FROM      sys.tables                 AS t
JOIN      sys.indexes                AS I ON  t.object_id     = i.object_id
JOIN      sys.partitions             AS p ON  i.object_id     = p.object_id
                                          AND i.index_id      = p.index_id
JOIN      sys.partition_schemes      AS s ON  i.data_space_id = s.data_space_id
JOIN      sys.partition_functions    AS f ON  s.function_id   = f.function_id
LEFT JOIN sys.partition_range_values AS r ON  f.function_id   = r.function_id
                                          AND r.boundary_id   = p.partition_number
WHERE     i.index_id <= 1;
```

Switching Partitions

No check constraint available

Partition boundaries must line up

- Single partition “in” table may not line up with defined partition boundaries of target table

Table Definition must match

- Data type
- Nullability

Statistics are not switched

- Update statistics for target post switch

Source Code Management

- Define table with partitioning
- Leave values empty
- Use a while loop to split partitions

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

Temporary Tables

Basic Facts

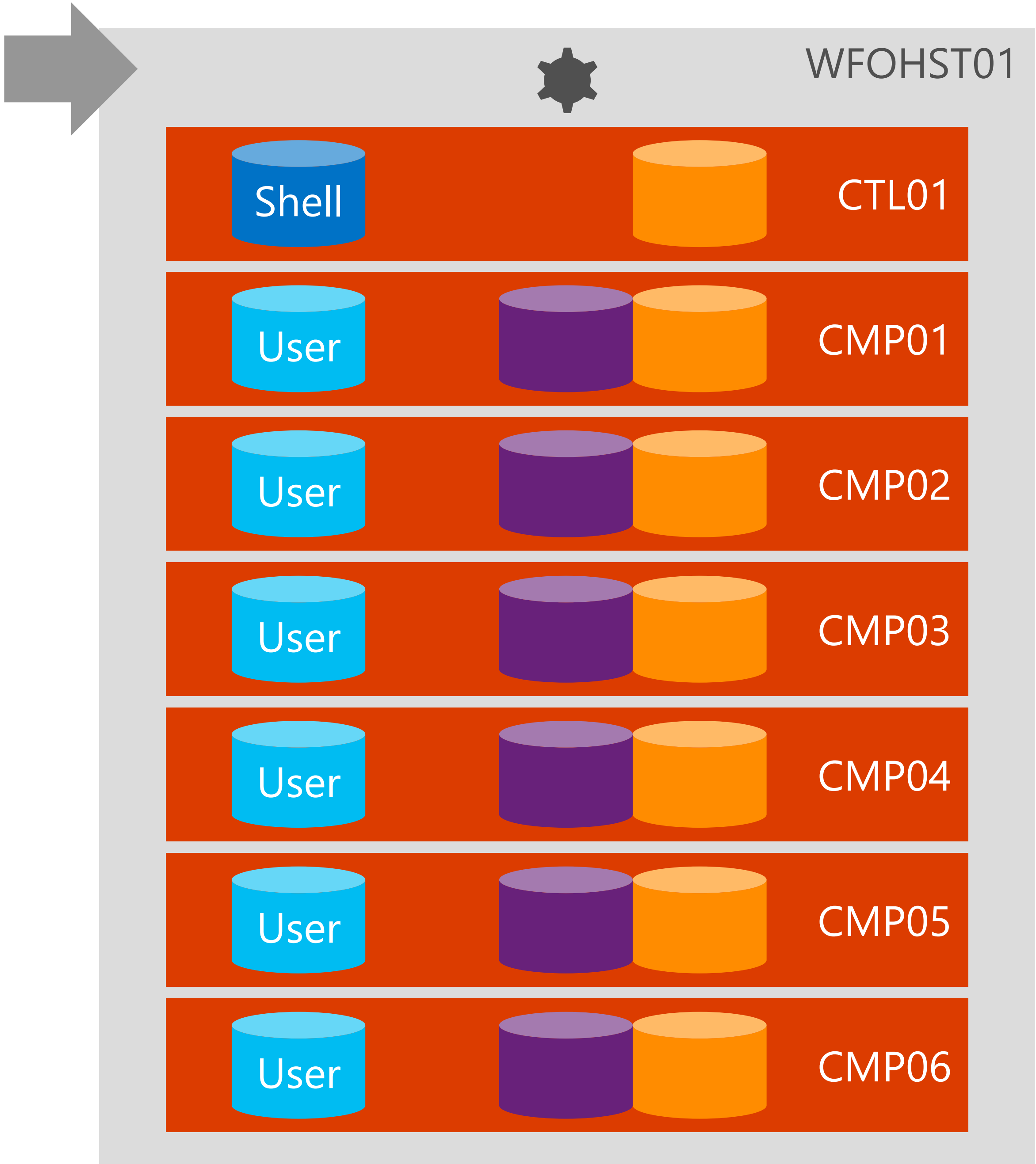
Can Temporary Tables be…	Answer
Replicated	Yes
Distributed	Yes
Rowstore	Yes
Columnstore	Yes
Created to span sessions? (i.e. global)	No
Supported by Statistics (Manual)	Yes
Supported by Statistics (Automatic)	No

Tempdb in PDW

- Tempdb
- pdwtempdb

Must specify

- #<table_name>
- WITH (LOCATION = User_DB)



Understanding Temporary Tables

Control node

- Tempdb used

Compute nodes

- pdwtempdb user defined Temp tables
- Tempdb Data Movement / QTables

Creating a Temporary Table

```
CREATE TABLE #Nums
WITH
( DISTRIBUTION = REPLICATE
, LOCATION = USER_DB
)
AS
WITH
  L0   AS(SELECT 1 AS c UNION ALL SELECT 1),
  L1   AS(SELECT 1 AS c FROM L0 AS A, L0 AS B),
  L2   AS(SELECT 1 AS c FROM L1 AS A, L1 AS B),
  L3   AS(SELECT 1 AS c FROM L2 AS A, L2 AS B),
  L4   AS(SELECT 1 AS c FROM L3 AS A, L3 AS B),
  L5   AS(SELECT 1 AS c FROM L4 AS A, L4 AS B),
  Nums AS(SELECT ROW_NUMBER() OVER(ORDER BY c)
AS n FROM L5)
SELECT n AS Number
FROM   Nums
WHERE  n <= @n;
```

Based on fn_nums by Itzik Ben-Gan

- Must use #
- Must specify Location=User_DB
- No global variables
- No Partitioning

Dropping a Temporary Table

```
IF OBJECT_ID('tempdb..#nums') IS NOT NULL
BEGIN
    DROP TABLE #nums
END
```

- Temp Tables cannot be renamed
- Automated deferred drop at end of session
- Best practice is to perform your own cleanup

Virtual Tables

Views

- Metadata Only – held in logical layer
- Read only
- Perform DML through base tables
- Built over permanent tables (not temp tables)
- Good architecture abstraction
- Good for enforcing optimised joins

Functions

Three primary forms of function

- Inline table functions
- Multi-statement table functions
- Scalar functions

User defined functions aren’t supported by PDW today

Common Table Expression (CTE)

Use cases
- Query simplification
- Create new objects
  - CTAS
  - CETAS

Limitations
- No DML
- No recursion

Using CTAS

General Guidance

- Think CTAS first
- If you can perform an operation with a CTAS then you probably should

Use Cases for CTAS

CTAS used to optimize

- Inserts
- Updates
- Delete

Use CTAS to

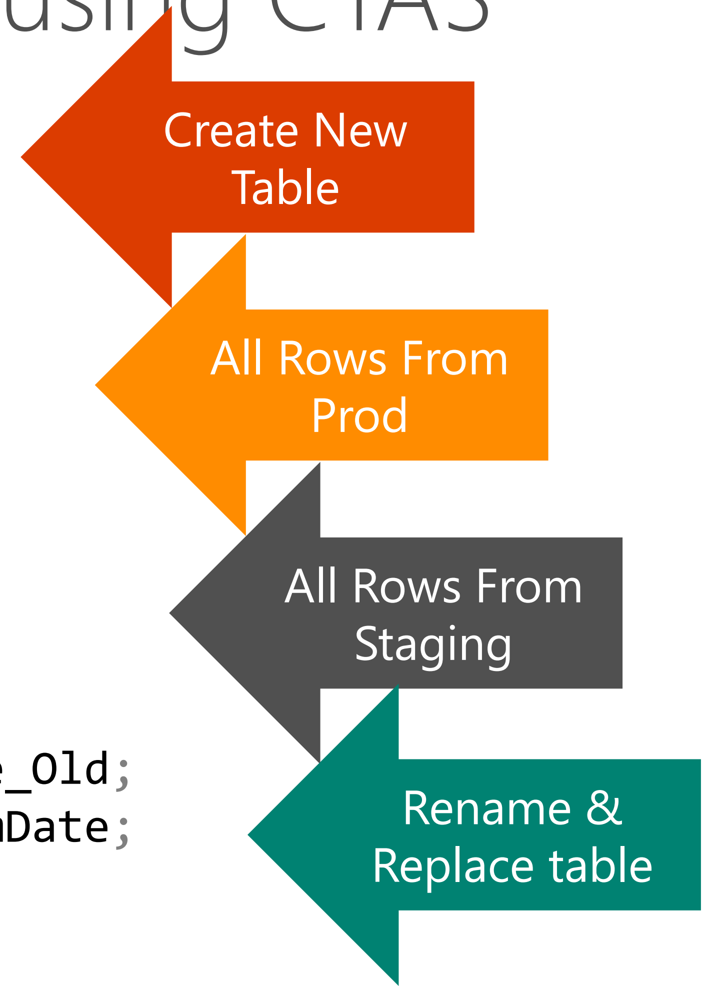
- Create Partitions
- Replace Merge
- Replace Index Rebuild

CTAS – Small Table "Insert"

Recreate the table using CTAS

```
CREATE TABLE dbo.DimDate_New
WITH (DISTRIBUTION = REPLICATE
, CLUSTERED INDEX (DateKey ASC)
)
AS
SELECT *
FROM   dbo.DimDate  AS prod
UNION ALL
SELECT *
FROM   dbo.DimDate_stg AS stg
;

RENAME OBJECT DimDate TO DimDate_Old;
RENAME OBJECT DimDate_New TO DimDate;
```



CTAS – Create Partition for Switch In

```
CREATE TABLE dbo.FactInternetSales_in
WITH (  Distribution=HASH (SalesOrderNumber)
     ,CLUSTERED COLUMNSTORE INDEX
  ,   PARTITION (OrderDateKey RANGE RIGHT FOR VALUES (20040101,20050101))
  )
AS
SELECT *
FROM   dbo.FactInternetSales tgt
WHERE  tgt.OrderDateKey >= 20040101
AND    tgt.OrderDateKey <  20050101
UNION ALL
SELECT *
FROM   dbo.stg_FactInternetSales stg
```

CTAS – Create Partition for Switch Out

```
CREATE TABLE dbo.FactInternetSales_out
WITH ( Distribution=HASH (SalesOrderNumber)
     ,CLUSTERED COLUMNSTORE INDEX
 ,  PARTITION (OrderDateKey RANGE RIGHT FOR VALUES (20040101,20050101))
 )
AS
SELECT *
FROM   dbo.FactInternetSales tgt
WHERE 1=2
```

Upsert example

```sql
SELECT
 CASE WHEN s.ProductAlternateKey IS NULL
      THEN p.ProductKey
      ELSE s.ProductKey
 END as ProductKey
,CASE WHEN s.ProductAlternateKey IS NULL
      THEN p.ProductAlternateKey
      ELSE s.ProductAlternateKey
 END as ProductAlternateKey
,CASE WHEN s.ProductAlternateKey IS NULL
      THEN p.EnglishProductName
      ELSE s.EnglishProductName
 END as EnglishProductName
,CASE WHEN s.ProductAlternateKey IS NULL
      THEN p.Color
      ELSE s.Color
 END as Color
FROM  DimProduct p
FULL JOIN stg_DimProduct s ON p.ProductAlternateKey = s.ProductAlternateKey
```

Compact Upsert statement

Maintainable Code but data movement will always exist for this query

Optimised Upsert

```sql
CREATE TABLE dbo.DimProduct_upsert
WITH (Distribution=HASH(ProductKey)
, CLUSTERED INDEX (ProductKey)
)
AS -- Update
SELECT      s.ProductKey
,           s.EnglishProductName
,           s.Color
FROM        dbo.DimProduct p
JOIN        dbo.stg_DimProduct s ON p.ProductKey = s.ProductKey
UNION ALL--Keep rows that do not exist in stage
SELECT      p.ProductKey
,           p.EnglishProductName
,           p.Color
FROM        dbo.DimProduct p
LEFT JOIN   dbo.stg_DimProduct s ON p.ProductKey = s.ProductKey
WHERE       s.ProductKey IS NULL
UNION ALL--Inserts
SELECT      s.ProductKey
,           s.EnglishProductName
,           s.Color
FROM        dbo.DimProduct p
RIGHT JOIN  dbo.stg_DimProduct s ON p.ProductKey = s.ProductKey
WHERE       p.ProductKey IS NULL
```

Works for Large Distributed Tables by eliminating data movement

Performance Optimisation Code is less maintainable

Delete example

```sql
CREATE TABLE dbo.DimProduct_upsert
WITH (Distribution=HASH(ProductKey)
, CLUSTERED INDEX (ProductKey)
)
AS -- Select Data you wish to keep
SELECT      p.ProductKey
,           p.EnglishProductName
,           p.Color
FROM        dbo.DimProduct p
RIGHT JOIN dbo.stg_DimProduct s
ON          p.ProductKey = s.ProductKey
```

Keeps only the data selected

In this case we keep Production Records that are either new or match Staging Records

Cautionary Note

```sql
DECLARE @d decimal(7,2) = 85.455
,@f float(24) = 85.455

CREATE TABLE result
(result DECIMAL(7,2) --defined as decimal (7,2)
)
WITH (DISTRIBUTION = REPLICATE)

INSERT INTO result
SELECT @d*@f

CREATE TABLE ctas_r
WITH (DISTRIBUTION = REPLICATE)
AS
SELECT @d*@f as result --defined as float

SELECT result,result*@d
from result

SELECT result,result*@d
from ctas_r
```

	result	second_result
1	7302.98	624112.6708

	result	second_result
1	7302.984	624113.1

Inserting into a table involves implicit type conversion

CTAS does not do this by default

Best Practice: Table Definition

Control Data Type and Nullability of columns from the outset and at source

- CAST in Select to control data type
- ISNULL around expression for NOT NULL

Best Practice: Table Definition

```
CREATE TABLE result
(result DECIMAL(7,2) --defined as decimal (7,2)
)
INSERT INTO result
SELECT @d*@f

CREATE TABLE ctas_r
AS
SELECT CAST(@d*@f AS DECIMAL(7,2)) as result --
defined as decimal(7,2)
```

Best Practice: Managed Workload

Problem:

- CTAS 5 large fact tables in a Union All to a Clustered Index Table may take too long and potential fill tempDB to capacity.

Solution:

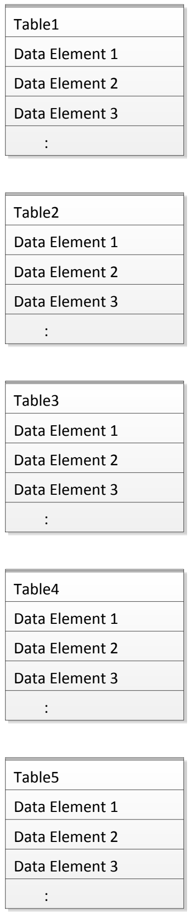
- Break up the workload into partition manageable chunks
- Use Partition switching logic

Managed Workload Example

Step 1 : CTAS

Step 2 : Switch Out

Step 3 : Switch In



Best Practices : Splitting Partitions

- Keep first and last partitions empty
  - Eliminates data movement
  - Metadata operation
- Use CTAS rather than Alter Table split
  - Alter Table could cause data movement
  - Data Movement runs parallel across appliance, serial across distributions
  - CTAS to new table and partition switch. 8X faster than Alter Table

Splitting Partitions via DDL (Slow)

DistA 20121201 20130115 20130215	DistB 20121201 20130115 20130215	DistC 20121201 20130115 20130215	DistD 20121201 20130115 20130215	DistE 20121201 20130115 20130215	DistF 20121201 20130115 20130215	DistG 20121201 20130115 20130215	DistH 20121201 20130115 20130215
	DistB 20130101	DistC 20130101	DistD 20130101	DistE 20130101	DistF 20130101	DistG 20130101	DistH 20130101
	DistB 20130201	DistC 20130201	DistD 20130201	DistE 20130201	DistF 20130201	DistG 20130201	DistH 20130201

Splitting Partitions with CTAS (Fast)

DistA
20130201
20130115
20130215

DistA
20121201
(Empty)

Step 1 : CTAS

Step 2 : Switch Out

Step 3 : Alter Table Split

(metadata operation)

Step 4 : Switch In

(metadata operation)

Limitations and Restrictions

- Cannot create a table with default constraints
  - Need to be applied afterwards
- Does not carry over indexes
- Does not carry over statistics

Create Table vs CTAS

Create Table
- Table is empty
- Rowcount set to 1000
- Data types from DDL
- Nullability from DDL
- Default constraints: yes
- Insert – partially parallel

CTAS
- Table populated
- Rowcount set to actual
- Data Type from Select
- Nullability from Select
- Default constraints: no
- BCP -Fully parallel

Using CRTAS

Create Remote Table As Select

- High performance data export to SQL Server
- Data streamed directly from compute nodes to SQL Server not via control node
- SQL Server must be on Infiniband network
- Counts as a query not as a load
- DMS Worker type = ‘Parallel Copy Reader’

Sample DDL

```
CREATE REMOTE TABLE
PDW_Student_export.dbo.Inventory_Student_XX
AT ('Data Source = 172.16.254.100,1433; User ID =
studentexport ;Password = PDWExp0rt3r;')
AS
SELECT *
FROM [dbo].[vSTUDENT_Inventory_CRTAS]
OPTION (LABEL = 'Inventory : CRTAS');
```

Implementation Limitations

The Remote Table Cannot

- Already exist
- Be partitioned during creation
- Be indexed during creation
- Be executed inside a transaction
- Use TOP(n) or Order by
- Use trusted authentication

External Tables

Creating an external table

Creates a Tabular structure over data

- Connection via Polybase

Requires configuration of additional DDL

- External data source – e.g. Hadoop
- External file format

Enables heterogeneous querying

Creating an external table

Specify column attributes

- Name
- Data type
- Collation
- Nullability

Specify additional properties

- Reject_type
- Reject_value
- Reject_sample_value

Create External Table As Select

- High performance data export via Polybase
- Target is any supported Polybase target
- Persists an External Table prior to export
- Counts as query not as a load
- DMS Worker Type = ExternalExport*

Table Design: Distribution Key

Choosing a Distribution Key

Three Basic Rules

- Avoid Data Skew
- Minimize Data Movement
- Provide Balanced Execution

Data Skew Defined

A table is skewed when:

“one or more distributions contain disproportionately more rows than the others”

In other words:

PDW performs only as fast as the slowest distribution

Root Causes of Data Skew

- Natural Skew
- Poor Hash
- Data Quality

Natural Skew

- Data volumes nucleate on a distribution key value:
- Item number for retailer
- Sales frequency of some items may be disproportionately higher than others e.g. cans of coke vs. Televisions

Natural Skew

Degree of skew

- Skew at the distribution level is where it matters most
- A few skewed items may be balanced out by the hash
- Natural skew may still hurt you if you query by the distribution key

Poor Hash from Poor Ratios

Distinct value : total row count is very high

- Null will always hash to the same distribution (1A)

distinct values : # distributions is very low

- 6 compute node appliance has 48 distributions
- StoreID would be a poor choice if there were only 50 stores
- Hash is simple no guarantee that the spread would over distributions

Data Quality

Disproportionate # NULLs in distribution key

- Inferred member (-1) is just as bad as NULL

Source System errors

- Re-sending duplicate data
- Re-using source system keys

PDW does not enforce uniqueness. This can become a problem if the key is heavily weighted to certain values

Avoiding Data Skew

Profile the Source

- Use Select Count / Group by to determine candidate column cardinality

Choose a column that

- Has a high number of distinct values
- Is defined as NOT NULL
- Doesn’t contain dominating values

Skew Guidance

Round up total number of distributions to nearest rack

- For a base unit round up to full rack of 8 or 9 Nodes

Multiply the total number of distributions by 10

- 8 compute nodes = 64x10 = 640 distinct values
- 9 compute nodes = 72x10 = 720 distinct values

A high hundreds – low thousands distinct value count will help future proof your design

Detecting Skew

- DBCC PDWSHOWSPACEUSED(<Table_Name>)

	ROWS	RESERVED_SPACE	DATA_SPACE	INDEX_SPACE	UNUSED_SPACE	PDW_NODE_ID	DISTRIBUTION_ID
1	8,922,051	237,840	237,792	8	40	201,001	1
2	8,940,913	238,416	238,384	8	24	201,001	2
3	8,893,038	237,072	237,064	8	0	201,001	3
4	8,896,245	237,136	237,096	8	32	201,001	4
5	8,917,352	237,712	237,704	8	0	201,001	5
6	8,890,298	237,008	236,968	8	32	201,001	6
7	8,895,185	237,072	237,048	8	16	201,001	7
8	8,891,385	237,072	237,016	8	48	201,001	8
9	8,887,681	236,944	236,880	8	56	201,002	1
10	8,879,381	236,688	236,656	8	24	201,002	2
11	8,890,218	237,008	237,000	8	0	201,002	3
12	8,880,976	236,816	236,752	8	56	201,002	4
13	8,881,714	236,752	236,712	8	32	201,002	5

Minimize Data Movement

- More important than skew
- Movement can re-introduce skew

Pay close attention to:

- Distributed tables that self-join
- Distributed tables that join to each other
- Outer joins – check for compatibility or convert

Where possible:

- Ensure the join contains the distribution key

Balanced Execution

Typical Column Distribution Pattern

- Column used in joins and group by

Anti-Pattern

- Distributing on a column used in where clause

Scenario: Choosing Distribution Key

Background

- Web analytics solution
- Lots of path analysis
- Typical analysis is by user
  - User_ID is commonly found in join criteria
  - Authenticated users have User_ID > 0
  - Anonymous users have User_ID = 0

Goal:
Choose the Optimal Distribution Key

Evaluate Root Cause Analysis

Minimize data movements

- Typical analysis is by User_ID
- Most queries are for authenticated users
- User_ID > 0

Balanced execution

- User_ID not typically used as a filter predicate (where clause)
- User_ID found in join criteria

Avoid data skew

- Data is heavily skewed by anonymous users
- User_ID=0

What solutions can you think of?

Solution: Two distribution keys

Authenticated Users

- Distribute by User_ID
- Have the analysts use this table

Anonymous Users

- Distribute by different key – Session_ID
- Use a View (union all) for whole fact analysis
- View will shuffle both tables
- Prior to shuffle data will be pre-aggregated and filtered to reduce move
- Whole table analysis rarely performed so movement cost acceptable

Note: solution fits query pattern

Table Design: Data Modelling

Scenario: Clickstream Analysis

[dbo].[fPageView]
[PageViewID] [bigint] NOT NULL,
[SessionID] [bigint] NOT NULL,
[SessionSequenceNum] [int] NOT NULL,
[PageID] [bigint] NOT NULL,
[DateID] [smallint] NOT NULL,
[PageViewDurationMinutes] [float] . . .

[dbo].[dPage]
[PageID] [bigint] NOT NULL,
[DomainName] [varchar](512) NOT NULL,
. . .
[PageName] [nvarchar](2000) NOT NULL,
[FullPagePath] [nvarchar](2000) NOT NULL,
[URIStem] [nvarchar](512) NULL ...

- Frequent joins between tables
- Frequent self joins on fPageView
  - Facilitates Session based path analysis

Warning!
Page_ID is heavily skewed to homepage in fPageView

Common Query Pattern – Clickstream

Most Popular “Triple” View Sequences

```
SELECT
. . .
FROM fPageView A
JOIN fPageView B
        ON  B.SessionID = A.SessionID
        AND A.PageID <> B.PageID
        AND B.SessionSequenceNum > A.SessionSequenceNum
JOIN fPageView C
        ON  C.SessionID = B.SessionID
        AND C.PageID <> B.PageID
        AND C.PageID <> A.PageID AND C.SessionSequenceNum > B.SessionSequenceNum
. . .
 JOIN dPage p1 ON p1.PageID = A.PageID
 JOIN dPage p2 ON p2.PageID = B.PageID
 JOIN dPage p3 ON p3.PageID = C.PageID
. . .
WHERE A.DateID BETWEEN 20100401 AND 20100630
```

Recommended Design Choices

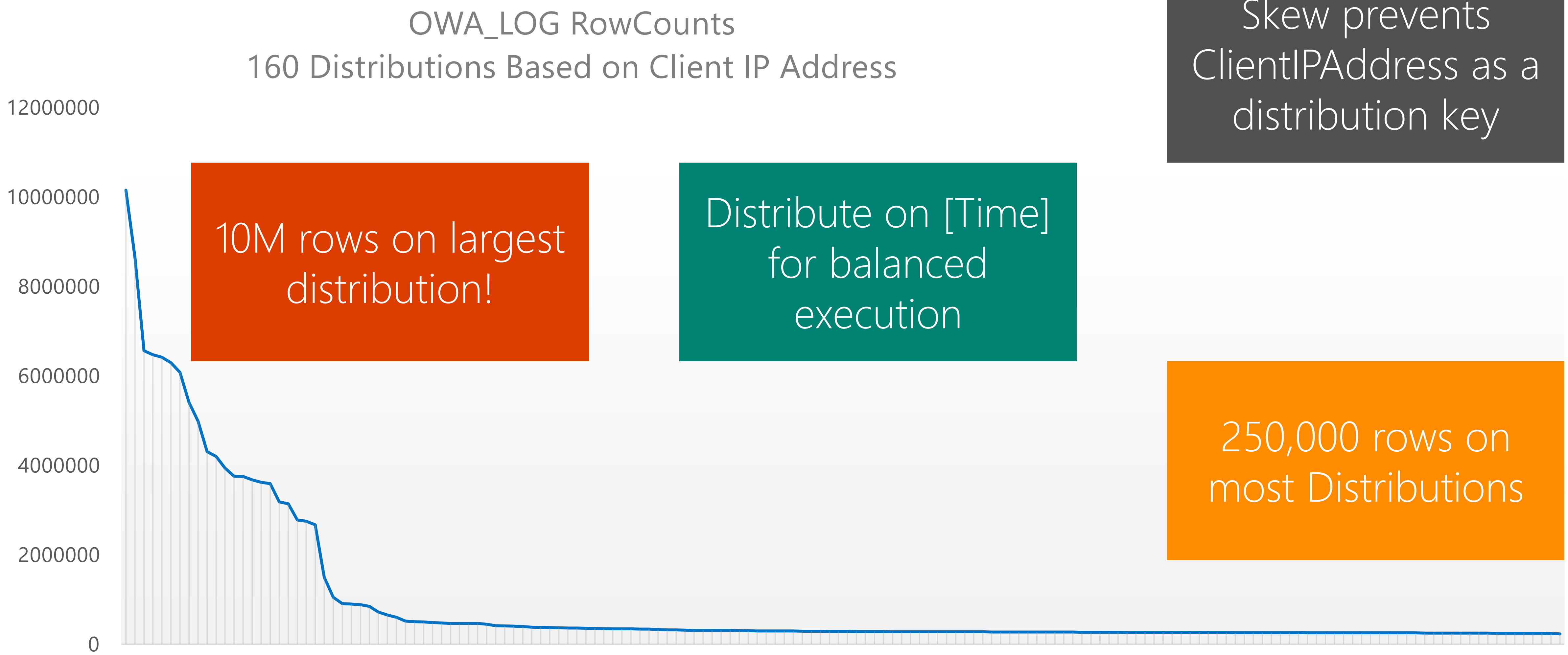
- dPage = replicated
- fPageView = distributed (SessionID)
  - PageViewID is nicely distributed, but useless for joins
  - With SessionID as distribution key, all fact-to-fact joins can be performed locally on the nodes: no shuffle
  - Joining to the replicated dPage dimension is local too
- Cluster and partition fPageView on DateID

Scenario 2: Network Security

dbo.[OWA_LOG]
[Date] datetime NULL, — Usually 1 date (latest) requested in query
[Time] time(7) NULL, — Random, to-the-second, large # values
[ServiceName] varchar(255) NULL, — Small number of values
[ServerName] varchar(255) NULL, — Small number of values
[ClientIPAddress] varchar(255) NULL, — Common join key; is it best?
. . .
[BytesSent] int NULL, — Significant skew
[BytesReceived] int NULL, — Significant skew
[TimeTaken] bigint NULL — Significant skew

OWA Log: 2TB
Joins to small fact
[DHCP_Log] on
[ClientIPAddress]

Trying Distribution on Client IP



10M rows on largest distribution!

Distribute on [Time] for balanced execution

Skew prevents ClientIPAddress as a distribution key

250,000 rows on most Distributions

Scenario 2: Network Security

dbo.[DHCP_LOG]
[Date] [datetime] NULL,
[Time] [time](7) NULL,
[EventID] [varchar](2) NULL,
[Description] [varchar](255) NULL,
[ClientIPAddress] [varchar](255) NULL,
. . .
[FullHostName] [varchar](255) NULL,
[DomainName] [varchar](255) NULL,
[IPAddressNbr] [bigint] NULL

- Usually 1 date (latest) requested in query
- Random, to-the-second, large # values
- Small number of values
- Small number of values
- Common join key; is it best?
- Significant skew
- Significant skew
- Significant skew

DHCP_Log: 36GB
Should we also distribute by [Time]?

Network Security – Running a Query

When [Time] is the dist key for both tables (OWA_LOG and DHCP_LOG):

- Join by ClientIPAddress required <u>both</u> tables to be shuffled
- Re-introduces the skew!

Solution: Make DHCP_LOG replicated

- Smaller of the two fact tables
- All joins performed locally
- no data skew and queries now distribution compatible
- Longer load time is a 1-time hit

Clickstream Revisited

[dbo].[fPageView]
[PageViewID] [bigint] NOT NULL,
[SessionID] [bigint] NOT NULL,
[SessionSequenceNum] [int] NOT NULL,
[PageID] [bigint] NOT NULL,
[DateID] [smallint] NOT NULL,
[PageViewDurationMinutes] [float] . . .

[dbo].[fCookiePageView]
[CookiePageViewID] [bigint] NOT NULL,
[PageViewID] [bigint] NOT NULL,
[SessionID] [bigint] NOT NULL,
[PageID] [bigint] NOT NULL,
[DateID] [smallint] NOT NULL,
[CookieID] [bigint] NOT NULL ...

- Introducing [fCookiePageView]
- Frequent joins on PageViewID
- Both tables distributed on SessionID

How do we avoid an expensive Shuffle?

Clickstream Revisited (continued)

- Adding a redundant join condition on SessionID makes the join operation distribution compatible!
- You’ll see dramatic increase in performance if you pay attention to data layout when developing queries
- Use views to guarantee redundant joins are used

Adding a Join Predicate

```
SELECT
. . .
FROM fPageView A
JOIN fPageView B
       ON B.SessionID = A.SessionID
       AND A.PageID <> B.PageID
       AND B.SessionSequenceNum > A.SessionSequenceNum
JOIN fPageView C
       ON C.SessionID = B.SessionID
       AND C.PageID <> B.PageID
       AND C.PageID <> A.PageID AND C.SessionSequenceNum > B.SessionSequenceNum
JOIN fCookiePageView cpv
       ON cpv.PageViewID = A.PageViewID
       AND cpv.SessionID = A.SessionID   --> needed for performance
 . . .
 JOIN dPage p1 ON p1.PageID = A.PageID
 JOIN dPage p2 ON p2.PageID = B.PageID
 JOIN dPage p3 ON p3.PageID = C.PageID
. . .
WHERE A.DateID BETWEEN 20100401 AND 20100630
```

Scenario: Master-Detail Dilemma

[dbo].[Orders]
[Order_ID] [bigint] NOT NULL,
[Customer_ID] [bigint] NOT NULL,
[Order_Date] [datetime2] NOT NULL,
[Channel_ID] [int] NOT NULL,
[ShippingType] [smallint] NOT NULL,
. . .

[dbo].[OrderDetails]
[Order_ID] [bigint] NOT NULL,
[Item_ID] [int] NOT NULL,
[Quantity] [int] NOT NULL,
[Price] [money] NOT NULL,
[Amount] [float] NOT NULL
. . .

- Two very large fact tables
- Analysis: Product behaviour aggregated by customer
- Therefore join is on Order but Group by is on customer
- Shuffle is required no matter which you pick!

Scenario: Master-Detail Solution

[dbo].[Orders]
[Order_ID] [bigint] NOT NULL,
[Customer_ID] [bigint] NOT NULL,
[Order_Date] [datetime2] NOT NULL,
[Channel_ID] [int] NOT NULL,
[ShippingType] [smallint] NOT NULL,
. . .

[dbo].[OrderDetails]
[Order_ID] [bigint] NOT NULL,
[Customer_ID] [bigint] NOT NULL,
[Item_ID] [int] NOT NULL,
[Quantity] [int] NOT NULL,
[Price] [money] NOT NULL,
[Amount] [float] NOT NULL
. . .

Join may not also be required!

- Flatten OrderDetails
- Add Customer_ID and use a composite join
- Joins and aggregation are distribution compatible

Microsoft