#MSMPPDWH

Azure SQL Data Warehouse

Tables and Querying

Microsoft

Agenda

- Table Geometries and Index structures
- MPP Engine Query Execution
- Cost-based Optimizer
- Control flow language constructs
- other T-SQL command and DB objects

Agenda

- Table Geometries and Index structures
- MPP Engine Query Execution
- Cost-based Optimizer
- Control flow language constructs
- other T-SQL command and DB objects

CREATE SCHEMA Command

```
CREATE SCHEMA schema_name [ AUTHORIZATION owner_name ] [;]
```

schema_name

- Is the name by which the schema is identified within the database.

AUTHORIZATION owner_name

- Specifies the name of the database-level principal that will own the schema. This principal may own other schemas, and may not use the current schema as its default schema.

Let's first talk about database SCHEMAs!
(Before we go to tables)

Because DB SCHEMAs are a container for DB objects!

SQL Data Warehouse runs the entire data warehouse workload within one database.

Cross database joins are not permitted.

SQL Data Warehouse expects all tables used by the warehouse to be stored within the one database.

Leverage user-defined schemas to provide the boundary previously implemented using databases

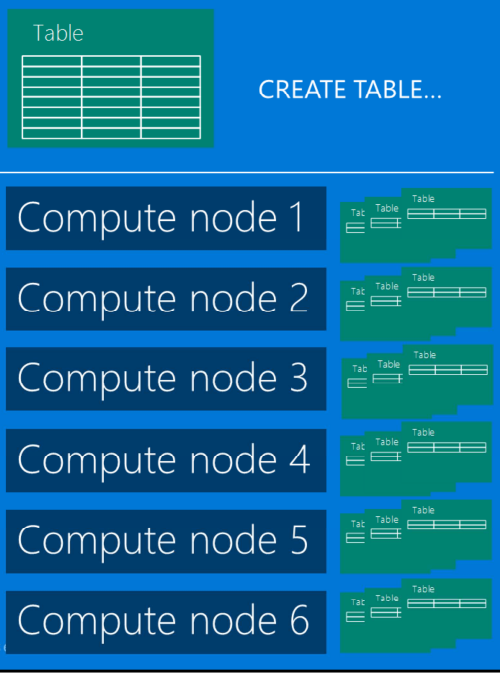
https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-develop-user-defined-schemas/

Table Geometries

A **distributed** table maintains only a single copy of the data, which is divided into separate physical tables on each compute node

- Each compute node has eight physical tables to store distributed data, known as **distributions**
- A six-node appliance has 48 distributions (6 x 8) in which to store records from a distributed table
- A 10-GB table on a six-node appliance requires **10 GB** of disk space
- The number of nodes is actually irrelevant when calculating distributed table sizing
- MPP Engine uses a **hash-** or a **round robin algorithm** to determine into which distribution to store a record. The hash algorithm is based on a value in the table column
- Typically, large fact tables are distributed

*Note that storage is not reflected in the diagram



https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-tables-distribute/

To HASH or ROUND_ROBIN?

Use HASH distributed tables when:

- For fact/detail tables if you **DO** have a common distribution key which **is used** in multi-distributed table joins
- If Full Table Scans do not provide acceptable performance

Use ROUND_ROBIN distributed tables when:

- On "Azure SQL Data Warehouse": instead of REPLICATED tables
- Multiple "primary/foreign key"-like joins are common
- When the data is an initial loading table

(Remember: Data movement will be most likely be involved in "multi distributed table joins", if at least one of them is ROUND_ROBIN distributed)

https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-develop-table-design/

CREATE TABLE Command

```
CREATE TABLE [database_name.][schema_name.]table_name
  (
    column_name <data_type>
        [ COLLATE Windows_collation_name ]
        [ NULL | NOT NULL ] }
        [ [ CONSTRAINT constraint_name ]
                DEFAULT constant_expression ] [ ,...n ]
  )
[ WITH ( <table_option> [ ,...n ] ) ]
[;]
```


Only Table Options is different from SQL Server.

CREATE TABLE Command

Temporary tables

```
CREATE TABLE [database_name.][schema_name.]#table_name
  (
    column_name <data_type>
        [ COLLATE Windows_collation_name ]
        [ NULL | NOT NULL ] }
        [ ,...n ]
  )
WITH ( LOCATION = USER_DB [, <table_option> [ ,...n ] ] )
[;]
```


Are very much like SMP SQL Server, simply add the # sign in front of the table name.

Local Temporary Tables are stored in the tempdb database. However, the LOCATION=USER_DB option is required when creating a temporary table

Global temporary tables (i.e. with the ## prefix) are not supported in the MPP DWH Engine.

Temporary tables are stored in the tempdb database. Temporary tables behave in the same manner as permanent tables unless explicitly stated otherwise. For example, within tempdb, user temporary tables are stored on the Compute nodes as distributed tables.
All temporary tables are local tables; they are only visible to the current session.

tempdb uses SQL Server minimal logging. This enables transactions to be rolled back.

Temporary tables are dropped from tempdb when:

- They are dropped explicitly with the DROP TABLE statement.
- A session is disconnected. Only temporary tables for the session are dropped.
- The appliance is shutdown.
- The Control node has a cluster failover.

CREATE TABLE Command (continued)

```
<table_option> ::=
    CLUSTERED COLUMNSTORE INDEX
    | CLUSTERED INDEX
       ( { index_column_name [ ASC | DESC ] } [ ,...n ] )
    | DISTRIBUTION = {
                       HASH ( distribution_column_name )
                     | ROUND_ROBIN
                     }
    | PARTITION
        ( partition_column_name RANGE [ LEFT | RIGHT ]
          FOR VALUES ( [ boundary_value [,...n] ] )
        )
```


Table limitations:

- All tables are created with SQL Server page compression on (not user-configurable)
- Minimize skew when choosing distribution key
- Iterative prototyping is critical
- Large fact tables are almost always implemented as distributed tables

Create Table Command - Example

```
CREATE TABLE myTable
        (
            id integer NOT NULL,
            lastName varchar(20),
            zipCode varchar(6)
        );
```

When distribution option is not specified, defaults to ROUND_ROBIN

Creating Tables - Limitations

Table limitations

- 2 billion tables per database
- Up to 1,024 columns per table
- Number of rows limited only by available storage
- Maximum bytes per row is 8,060

The "maximum bytes per row" is allowed for tables with varchar columns that cause the total defined table width to exceed 8,060 bytes.

- The actual width of all columns must still fall within the 8,060 byte limit.

Current Limitations of Data Types

Most Scalar data types supported by
SQL Server are supported by the MPP DWH

Main exceptions

- Text (and related BLOB data types, including `xxxxxx(MAX)`)
- XML
- SQL variant
- Timestamp
- System and CLR UDTs

IDENTITY columns and PK/FK constraints not supported

Collations are fully supported, similar to SQL Server, on the Column Level

- Default: SQL_Latin1_General_CI_AS_KS_WS
  also possible: Latin1_General_100_CI_AS_KS_WS (on the DB level)

SQL Server Data Types	PDW
bigint	✓
binary	✓
bit	✓
char/nchar	✓
date, time	✓
datetime	✓
datetime2	✓
datetimeoffset	✓
decimal	✓
float	✓
geography / geometry	
hierarchyid	
image	
int	✓
money	✓
numeric	
real	✓
smalldatetime	✓
smallint	✓
smallmoney	✓
sql_variant	
sysname	
text / ntext	
timestamp	
tinyint	✓
uniqueidentifier	
varbinary	✓
varchar / nvarchar	✓
xml	

Text, Image, XML and all xxx(MAX) data types are NOT commonly used in a data warehouses.
➔they are not supported

Sort Order: Default for columns is what is default on the DB Level and the options are on the slide.
➔On the column level, any SQL Server supported Windows Collation be used

The MPP Engine is generally case in-sensitive:
- Applies to all character-based data types

https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-tables-data-types/

Constraints

Primary keys and unique indexes are *not* supported

Foreign constraints are *not* supported

Default constraints are supported

- CREATE TABLE
- ALTER TABLE
- Must be a literal value or a constant

Unique constraints are not supported

Nonclustered indexes:
- Compressed (like all MPP DWH data)
- Can be multi-column

Additional Table Geometry

Physical table structures

- Heap
- Clustered Index
- Clustered Columnstore Index

Additional table geometries

- Partitions
- Non-clustered indexes

ALTER TABLE Command

Modify/Add/Drop Column

Cannot be modified or dropped

- Distribution column
- Column included in index
- Partition column

REBUILD (all or specified partitions)

SPLIT/MERGE/SWITCH Partition

ALTER TABLE

```
ALTER TABLE [ database_name . [schema_name ] . | schema_name. ] source_table_name
{
    ALTER COLUMN column_name
        {
            type_name [ ( precision [ , scale ] ) ]
            [ COLLATE Windows_collation_name ]
            [ NULL | NOT NULL ]
        }
    | ADD { <column_definition> | <column_constraint> FOR column_name} [ ,...n ]
    | DROP { COLUMN column_name | [CONSTRAINT] constraint_name } [ ,...n ]
    | REBUILD [ PARTITION = { ALL | partition_number } ]
    | { SPLIT | MERGE } RANGE (boundary_value)
    | SWITCH [ PARTITION source_partition_number
        TO target_table_name [ PARTITION target_partition_number ]
}
[;]
```


You cannot move partitions between distributed and replicated tables.

ALTER TABLE

```
<column_definition>::=
{
    column_name
    type_name [ ( precision [ , scale ] ) ]
    [ <column_constraint> ]
    [ COLLATE Windows_collation_name ]
    [ NULL | NOT NULL ]
}

<column_constraint>::=
    [ CONSTRAINT constraint_name ] DEFAULT constant_expression
```

TRUNCATE TABLE Command

Works with distributed tables

Works with permanent or temporary tables

Not allowed

- with EXPLAIN
- within user-defined transaction

Performance

- For a replicated table, TRUNCATE TABLE is executed in parallel across the compute nodes
- For a distributed table, TRUNCATE TABLE is executed in parallel across the compute nodes, and serially across the distributions within each compute node

#MSMPPDWH

Azure SQL Data Warehouse

Partitioning

Microsoft

Partitioning Distributed Tables

- Distributed tables are already segmented by distributions
- Will further partition rows within a distribution, based on a partition function
- Enables for operations efficiency when adding, loading, dropping, and switching partitions
- Good for fast loading of an unused partition and then switching it in after loading

EXAMPLE:
hash on product id
partition on order date
query sales for each product filtered on a timespan

https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-develop-table-partitions/

Partitioned Table Usage

- Same basic rules and guidelines as SQL Server SMP
- Use partition switch for large inserts, updates, and deletes
- Use metadata queries to explore partitions
  - Number of partitions and boundary values
  - Partitioning column name

Partitioned Table Management

- Provides the ability to create, merge, split, and switch partitions
- Allows DBAs to administer large tables in smaller chunks for loading, archiving, sliding windows, and more
- Optimizes the loading of large tables
- Enables re-indexing of smaller partitions
- Enables piecemeal data reorganization

https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-tables-partition/

Create Distributed Table With Partitions

```
CREATE TABLE myTable
  (
    id integer NOT NULL,
    lastName varchar(20),
    shipdate datetime
  )
  WITH
    ( DISTRIBUTION = HASH (id),
      CLUSTERED INDEX (shipdate),
      PARTITION (shipdate  RANGE RIGHT FOR VALUES
                  ( '1992-01-01', '1992-02-01', '1992-03-01',
                    '1992-04-01', ... )
                )
    );
```


https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-tables-partition/

Partitioned Table with a Clustered Index

- May improve range queries
- Can hinder data load times more significantly on an MPP system than a SQL Server SMP system
- Page fragmentation can hinder query performance

➔FRAGMENTATION: This gets less important YoY, especially since In-Memory technologies hold relevant data mostly in RAM and machines have more and more RAM! (Remember RAM is cheap these days!)

Partitioned table clustered index:

- Guidelines apply to both replicated and distributed table
- Can use CTAS (discussed later in this lesson) to reorganize a table
- Despite the drawbacks, partitioned tables are still commonly used in MPP Database schemas

Partitioned Table with a Clustered Columnstore Index

- May improve range queries
- Column oriented Store (not row oriented)
  - ➔ Lowest memory footprint if only a few columns of a table are used
  - ➔ Best compression Ratio

First choice for large FACT tables!

- Query speed may get significantly enhanced when table is loaded sorted

Partitioned Table with a Non-Clustered Index

- Usually not recommended
- Add only with great care and a lot of testing
- Can be potentially used for better concurrency
- Can cause lots of random I/O
  - Heap tables promote more sequential I/O, which generally works best in the MPP world as well as under concurrent workloads

➔SEQUENTIAL I/O vs RANDOM I/O: This gets less important YoY, especially since In-Memory technologies hold relevant data mostly in RAM and machines have more and more RAM! (Remember RAM is cheap these days!)

Partitioned table with a nonclustered index:

- Guidelines apply to both replicated and distributed tables
- Consider an iterative design approach to test load and query performance

Partitioning

Partition for manageability (maintenance)

- Typically on a date key (or integer surrogate)
- Typically same as clustered index key
- SWITCH partitions:
  - OUT for fast delete of history
  - IN to modify or add a specific historical slice
  - IN if data that is being loaded is used for maintaining a separate copy of load (e.g., for feeding DEV/TEST/DR – "dual loading")

"dual loading": will be explained later in the course…

Similar considerations here as in SMP SQL designs:

-

- Place clustered key and partition key on the same column.
- Use partitions to switch out old data based on date.
- May avoid using clustered index if partitioning is fine-grained enough to limit scans – keep as a heap instead.

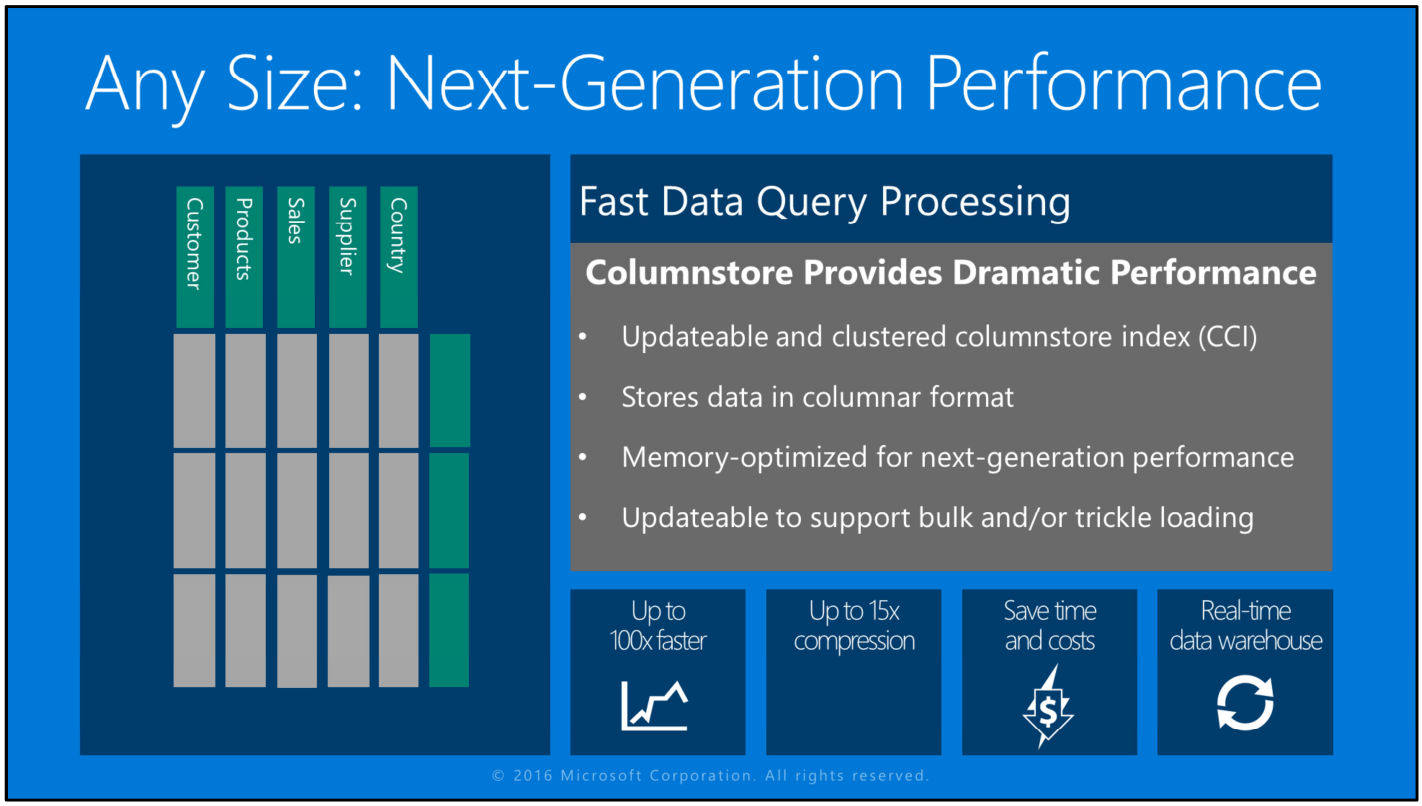
Note: For more information on SQL Server partitioning, see http://technet.microsoft.com/en-us/library/cc966457.aspx

#MSMPPDWH

Azure SQL Data Warehouse

Indexing

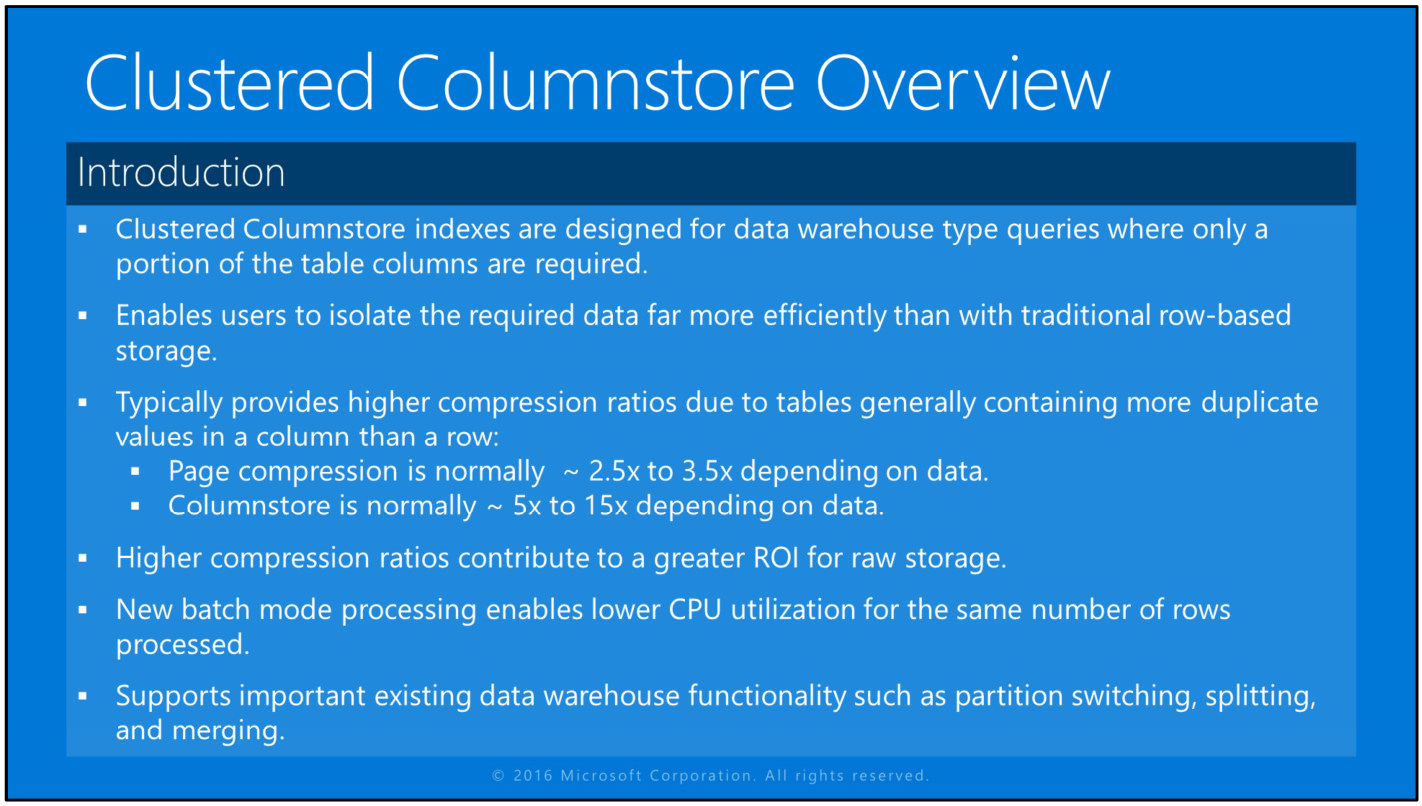
Microsoft



Remember that slide? We refer here back to the positioning slide deck.

CCI is still a little new to most customers! – so we explain it in details. – It was also new in SQL Server 2014!!
➔ CCI was first in MPP SQL, already 2012 an came then to SQL Server 2014!

In SQL Server 2012 there was a only a "NON-Clustered" version of the "Column Store" available which is not comparable to "Clustered Column Store" (CCI)!
"NON-Clustered Column Store" is not available in MPP SQL! (It was Read Only and so was using it a management overhead. Such a concept is not applicable for PB scale.)



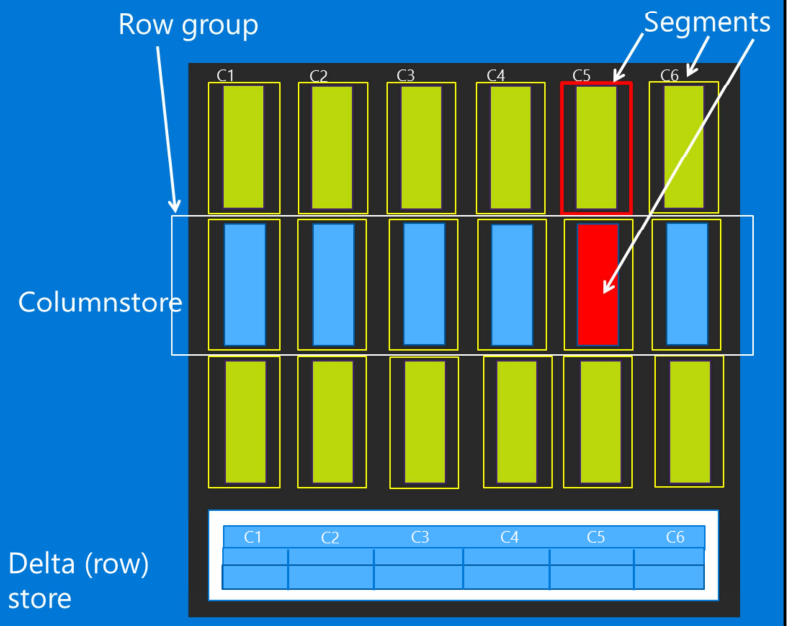
CCI is still a little new to most customers! – so we explain it in details. – It was also new in SQL Server 2014!!
➔ CCI was first in MPP SQL already 2012 an came then to SQL Server 2014!

In SQL Server 2012 there was a only a "NON-Clustered" version of the "Column Store" available which is not comparable to "Clustered Column Store" (CCI)!
"NON-Clustered Column Store" is not available in MPP SQL! (It was Read Only and so was using it a management overhead. Such a concept is not applicable for PB scale.)

Clustered Columnstore Overview

Terminology

- Clustered columnstore index is comprised of two parts:
  - **Columnstore**
  - **Deltastore**
- Data is compressed into **segments**
  - Ideally ~1 million rows (subject to system resource availability)
- A collection of segments representing a set of entire rows is called a **row group**
- The minimum unit of I/O between disk and memory is a **segment** (red block is a single segment)
- Execution **batch model** (as opposed to traditional row mode) moves multiple rows between iterators:
  - ~ 1000 Rows
- **Dictionaries** (primary and secondary) are used to store additional metadata about segments

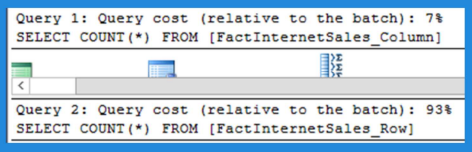


Clustered Columnstore Overview

More than compression – batch mode

- Since SQL 2012 **batch mode** processing is available to handle batch-at-a-time in addition to a row-at-a-time
  - SQL 2008 and before only had **row** processing
- Typically batches of about 1,000 rows are moved between iterators
  - Significantly less CPU is required due to the average number of instructions per row decreasing
- Batch mode processing is only available for some operators
  - Hash Join/Aggregate are supported
  - Merge Join, Nested Loop Join, and Stream Aggregate are not supported

SELECT COUNT(*) FROM FactInternetSales_Column	352 ms
SELECT COUNT(*) FROM FactInternetSales_Row	6704 ms



Row mode scan example

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	12079600
Actual Number of Batches	0
Estimated I/O Cost	35.1579
Estimated Operator Cost	41.8018 (92%)
Estimated CPU Cost	6.64386
Estimated Subtree Cost	41.8018

Batch mode scan example

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	ColumnStore
Actual Number of Rows	12079600
Actual Number of Batches	13463
Estimated Operator Cost	0.667511 (19%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.664386

It is possible for SQL Server to use the ColumnStore to scan the data but execute in Row Mode

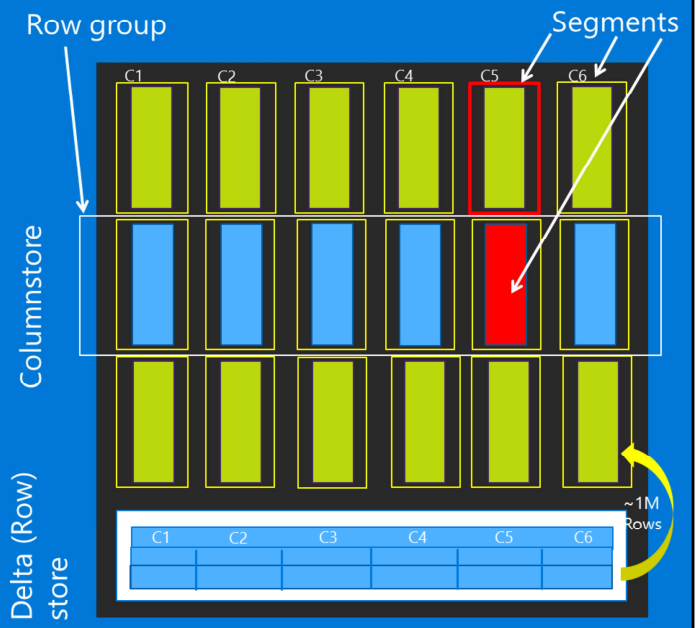
SQL may Estimate Batch but Actual is Row, (Memory or Threads – no Batch in MAXDOP 1) but will never estimate Row but Actual is Batch

Queries were executed against Cold Cache

Clustered Column Store Index Design

How DML is supported

- Changes to data can be applied directly to a clustered columnstore index
- INSERTs are added to the deltastore table
  - NOTE: The deltastore table is a **Page Compressed Heap**
- DELETEs from columnstore are logical; data is not physically removed until **REBUILD** is issued
  - DELETE from deltastore is physical as it is row based
- UPDATE is INSERT + DELETE
- Deltastore is automatically converted to columnStore at ~1M rows by background "Tuple Mover" process
  - Can also be forced with **REORGANIZE** at ~1M rows
- Converting deltastore to columnstore with **REORGANIZE** is an **ONLINE** operation
- **ADD** / **DROP** / **ALTER COLUMN** is supported
  - Partition switching also supported



- REBUILD is an OFFLINE operation, typically performed as a maintenance task

Clustered Columnstore Index Design

DMV Show Data Change

- A single record will get INSERTED into a table with clustered columnstore index
- Screenshots taken from DMV **sys.pdw_nodes_column_store_row_groups** (subset of total rows returned)
- Before row inserted:

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes	pdw_node_id
1714105147	1	1	0	NULL	3	COMPRESSED	1756	0	35512	201001
1714105147	1	1	0	NULL	3	COMPRESSED	1921	0	38816	201002

- After single row inserted (deltastore has been created and single row represented)

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes	pdw_node_id
1714105147	1	1	1	72057594148356096	1	OPEN	1	NULL	NULL	201001
1714105147	1	1	0	NULL	3	COMPRESSED	1756	0	35512	201001
1714105147	1	1	0	NULL	3	COMPRESSED	1921	0	38816	201002

- After REBUILD
  - Segment row count increased by 1

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes	pdw_node_id
1714105147	1	1	0	NULL	3	COMPRESSED	1757	0	35544	201001
1714105147	1	1	0	NULL	3	COMPRESSED	1921	0	38816	201002

  - REORGANIZE only moves "Closed" delta store segments into a "Compressed" status
  - REBUILD affects entire index (or entire partition index)

Clustered Columnstore Index Design

Columnstore data movement

- The state_description field has three states: **COMPRESSED, OPEN,** and **CLOSED**
  - COMPRESSED represents a row group that is stored in columnstore format
  - OPEN represents a deltastore that is accepting new rows
  - CLOSED represents a full deltastore ready for REORGANIZE

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
1618104805	1	1	0	NULL	3	COMPRESSED	1048576	0	355704
1618104805	1	1	1	NULL	3	COMPRESSED	951424	0	364576
1618104805	1	1	2	72057594047037440	2	CLOSED	1048576	NULL	NULL
1618104805	1	1	3	72057594047102976	1	OPEN	7424	NULL	NULL

- When inserting 102,400 rows or **more** in a single batch into a columnstore index distribution, the data will compress automatically

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
1266103551	1	1	0	NULL	3	COMPRESSED	102400	0	1551440

- When inserting 102,399 or **less** rows in a single batch, the data will be stored in the delta store

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
1266103551	1	1	0	72057594047234048	1	OPEN	102399	NULL	NULL

- The actual maximum number of rows per delta store is 1,048,576 at which point it is CLOSED
  - This is also the ideal segment size that SQL Server will try to create when first building a columnstore index from a table
  - When Index Build encounters memory pressure, DOP is reduced first and then segment size is reduced
- Only the REBUILD statement can compress a delta store that is not in the CLOSED state
  - Neither REORGANIZE or the Tuple Mover process will have any effect on an OPEN delta store

A single INSERT batch that reaches and exceeds the 1,048,576 row limit will fill the existing Delta Store and the remaining rows will be put into a new Delta Store

Clustered Columnstore Index Design

Columnstore is partition friendly

Consider a table with a single partition range of 100 with two partitions

- INSERT the partition key values *50* and *150* into the table

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
1682105033	1	1	0	72057594048872448	1	OPEN	1	NULL	NULL
1682105033	1	2	0	72057594048937984	1	OPEN	1	NULL	NULL

- REBUILD the index on partition 1

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
1682105033	1	1	0	NULL	3	COMPRESSED	1	0	80
1682105033	1	2	0	72057594048937984	1	OPEN	1	NULL	NULL

- INSERT three more records into partition 1 (Note partition 2 is still in deltastore form also)

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
1682105033	1	1	0	NULL	3	COMPRESSED	1	0	80
1682105033	1	1	1	72057594049134592	1	OPEN	3	NULL	NULL
1682105033	1	2	0	72057594048937984	1	OPEN	1	NULL	NULL

- REBUILD the entire index (as opposed to the single partition above)

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
1682105033	1	1	0	NULL	3	COMPRESSED	4	0	80
1682105033	1	2	0	NULL	3	COMPRESSED	1	0	80

Note: When attempting a partition **MERGE** or **SPLIT** operation on a **populated** partition, be aware that a table with a columnstore index will return an error and fail to execute whereas a traditional row-based clustered index/heap will perform the operation successfully. **MERGE** and **SPLIT** operations on empty partitions in a Clustered Columnstore Index table will execute successfully.

A single INSERT batch that reaches the 1,048,576 row limit will fill the remaining Delta Store and the remaining rows will be put into a new Delta Store

Clustered Columnstore Metadata

SQL DMVs extended for MPP SQL

Existing SQL 2012/2014 DMVs had been extended with the column "pdw_node_id" and got a new name

sys.pdw_nodes_column_store_segments

ent_id	version	encoding_type	row_count	has_nulls	base_id	magnitude	primary_dictionary_id	secondary_dictionary_id	min_data_id	max_data_id	null_value	on_disk_size	pdw_node_id
	1	1	1843	0	211	1	-1	-1	214	606	-1	2408	201003
	1	1	1843	0	20010709	1	-1	-1	20010712	20040730	-1	3984	201003
	1	1	1843	0	20010721	1	-1	-1	20010724	20040811	-1	3984	201003

sys.pdw_nodes_column_store_dictionaries

partition_id	hobt_id	column_id	dictionary_id	version	type	last_id	entry_count	on_disk_size	pdw_node_id
72057594142982144	72057594142982144	6	0	1	1	5	3	68	201001
72057594142982144	72057594142982144	7	0	1	1	8	6	80	201001
72057594142982144	72057594142982144	8	0	1	1	11	9	92	201001
72057594142982144	72057594142982144	9	0	1	3	834	831	9908	201001
72057594142982144	72057594142982144	10	0	1	1	9	7	84	201001

sys.pdw_nodes_column_store_row_groups (shown earlier in INSERT example)

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes	pdw_node_id
1666104976	1	1	0	NULL	3	COMPRESSED	1939	0	38992	201002
1746105261	1	1	0	NULL	3	COMPRESSED	1949	0	39272	201002
1730105204	1	1	0	NULL	3	COMPRESSED	1933	0	38992	201002

DELETE Row Count

- **sys.indexes** and **sys.index_columns** have also been modified for columnstore information

Clustered Columnstore Overview

Conclusion

- **Three factors contribute to exceptional query performance:**
  1. **Columnstore index segment elimination**
  2. **Batch mode execution**
  3. **Highest Compression → Lowest I/O**
- Clustered Columnstore indexing is significantly advanced from the SQL Server 2012 variation
  - Single copy of data
  - Full DML Supported
  - Query processor enhancements
- Be aware of the circumstances in which data is automatically compressed (rather than delta stored)

Remember:

CCI is still a little new to most customers! – so we did explain it in details. – It was also new in SQL Server 2014!!

→ CCI was first in MPP SQL already 2012 an came then to SQL Server 2014!

In SQL Server 2012 there was a only a "NON-Clustered" version of the "Column Store" available which is not comparable to "Clustered Column Store" (CCI)!

"NON-Clustered Column Store" is not available in MPP SQL!

Clustered Index vs. CCI vs. Heap Table

Clustered Index (CI)

- Introduces a sort step in table loads (loads slower than heap)
- Requires periodic maintenance to defragment (CTAS)
- Cluster fact tables on same column used for partitioning for optimal sequential I/O

Clustered Columnstore Index (CCI)

- Offers highest compression
- Optimized for Data Warehouse House (DWH) Workload

Heap Table

- Have fastest rate for inserts/loads ➔ e.g. for "store only" tables

Additional Considerations

- Minimize secondary (non-clustered) indexes with MPP DWH to reduce random I/O

➔FRAGEMNTATION: This gets less important YoY, especially since In-Memory technologies hold relevant data mostly in RAM and machines have more and more RAM! (Remember RAM is cheap these days!)

➔"store only" tables: these are typically logging tables or append only tables – otherwise have HEAPS a disadvantage for SELECTs

Clustered table

- A clustered index is the table
-

Heap table

- Provides for the fastest data loading

Non-Clustered Indexes

Use "judiciously"

- Will become fragmented with DML
  - Use Alter Index to defrag
- Will affect performance of DML operations
- Will take disk space

"judiciously": The MPP DWH Engine is optimized for FAST SCANs, so the need to "avoid SCANs as muss as possible", as in SMP SQL Server, is not a goal for optimization and one of the most important reasons why these indices are used in SQL Server SMP.

➔For that reason, there is no longer a need to use NC-Indices!!

Nonclustered indexes:

- Compressed (like all MPP DWH data)
- Can be multi-column

#MSMPPDWH

Azure SQL Data Warehouse

Statistics

Microsoft

What are Statistics?

Database objects!

- Contain statistical information about the distribution of values in a column
  - The statistics object includes a histogram to show the distribution of values in the first column
- Multi-column statistics can also be generated
  - These also hold density information i.e. the correlation of values between columns

https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-develop-statistics/

Statistics

Statistics on compute nodes

- The MPP Engine uses automatic statistics settings on compute nodes
- Compute node statistics use standard SQL Server sampling
  - Updated after table changes by 20 percent
- Can use compound (covering) statistics

Statistics on control node

- The MPP Engine uses statistics on control node to improve performance by using cardinal estimation
- Do **not** updated automatically!

Since statistics on control node do not get updated automatically, it is important to maintain statistics on a regular basis

Statistics in the MPP DWH Engine:
- Stats do not get recalculated in MPP DWH Engine
- Multicolumn (covering) statistics must be always manually created (also in “SQL Server SMP”)

Multicolumn statistics

If your joins are on multiple key columns, multicolumn stats help avoid mistaken "nested loop" plans on SQL Server node

Example:

- A long-running query step involved multicolumn join key and was experiencing lots of reads
- SQL query plan (estimated) showed nested loop and a very small number of estimated result rows from the join
- Actual query had millions of rows running through a nested loop operation

Statistics

Syntax:

```
CREATE STATISTICS statistics_name
    ON [ database_name.[schema_name]. | schema_name. ]table_name
    ( column_name  [ ,...n ] )
    [ WHERE <filter_predicate> ]
    [ WITH {
           FULLSCAN
           | SAMPLE number PERCENT }
    ][;]
```

```
<filter_predicate> ::=
    <conjunct> [AND <conjunct>]

<conjunct> ::=
    <disjunct> | <comparison>

<disjunct> ::=
        column_name IN (constant ,…)
```

```
<comparison> ::=
        column_name <comparison_op> constant

<comparison_op> ::=
    IS | IS NOT | = | <> | != | > | >= | !>
               | < | <= | !<
;
```

What are Statistics (in practice)

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	PK_IDx1	May 27 2014 4:37PM	122200000	492585	106	0	20	NO	NULL	122200000

	All density	Average Length	Columns
1	0,009433962	13	Machineid
2	2,030107E-06	20	Machineid, UtcTime

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	3	0	126768,4	0	1
2	7	0	198711,3	0	1
3	54	0	72439,07	0	1
4	55	0	54329,3	0	1
5	167	0	1316059	0	1
6	401	0	1499142	0	1
7	430	0	1906487	0	1
8	450	0	1811969	0	1
9	482	0	1906735	0	1
10	510	0	1258753	0	1
11	511	0	1301671	0	1
12	674	0	1417027	0	1

Best Practices : Statistics

Background

- Two levels of statistics exists in the MPP DWH Engine
  - SQL level – on Compute Nodes – maintained automatically
  - MPP level – cumulative statistics on Control Node's Shell DB for cost based optimization
- MPP level statistics is not maintained/created automatically

Goal

- Updated statistics for best cost based optimization

Best Practices : Statistics

Solution - Two options

1. Create statistics for all columns used in JOINs, GROUP BY, WHERE
   - Smart approach, but may not be feasible in all scenarios

2. Create statistics for all columns and all tables
   - Easy, but implies overhead

#MSMPPDWH

Azure SQL Data Warehouse

CREATE TABLE AS SELECT

Microsoft

Create Table as Select (CTAS)

Creates a new table based on a SELECT statement

Can be used to change:

- Distributed table
- The distribution column
- The clustered index or changes to a heap
- Partitioning

Can be use to defrag tables

CTAS:
- Always goes into a new, not yet existing table
  - Allows for minimal logging
  - Can later rename an object to rebind to previous dependencies (e.g., view)
- Can be used to replace SQL's standard INSERT, UPDATE, and DELETE statements
  - Especially with large data operations
- Can transform between replicated and distributed tables

https://azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-develop-ctas/

CREATE TABLE AS SELECT – Syntax

```
CREATE TABLE [ database_name . [ schema_name ] . |
schema_name. ] table_name
        [ ( { column_name } [ ,...n ] ) ]
    WITH ( DISTRIBUTION =
            { HASH( distribution_column_name )
            | ROUND_ROBIN }
            [ , <CTAS_table_option> [ ,...n ] ]
    )
    AS <select_statement> [;]
```


CTAS syntax – CTAS Table options is next slide

CREATE TABLE AS SELECT – Syntax (cont.)

```
<CTAS_table_option> ::=
    LOCATION = USER_DB
    | CLUSTERED COLUMNSTORE INDEX
    | CLUSTERED INDEX
        ( { index_column_name [ ASC | DESC ] } [ ,...n ] )
    | PARTITION( partition_column_name
            RANGE [ LEFT | RIGHT
            FOR VALUES ( [ boundary_value [,...n] ] ) ) ]

<select_statement> ::=
    [ WITH <common_table_expression> [ ,...n ] ]
    SELECT <select_criteria>
```

CTAS Compared to SQL's Insert/Select

Insert/Select

- Runs in parallel across nodes but "sequentially" across distributions
- Each distribution is run separately to provide transaction safety across the entire appliance
- Runs within a SQL Server transaction
- Full transaction rollback occurs in event of failure

Because CTAS runs in parallel, it is dramatically faster!

CTAS Compared to SQL's Insert/Select (cont.)

CTAS

- Runs fully in parallel across nodes across distribution
- Minimal logging

Performance

- Roughly eight times better performance compared to INSERT/SELECT and the APS Appliance
- In SQL DW, performance depends highly on the DWU setting

CREATE TABLE statement

- Destination table can be used with partition switching if exists

Because CTAS runs in parallel, it is dramatically faster!