

Columnstore Index Structure

Row Groups & Segments

Segment

A segment contains values for one column for a set of rows

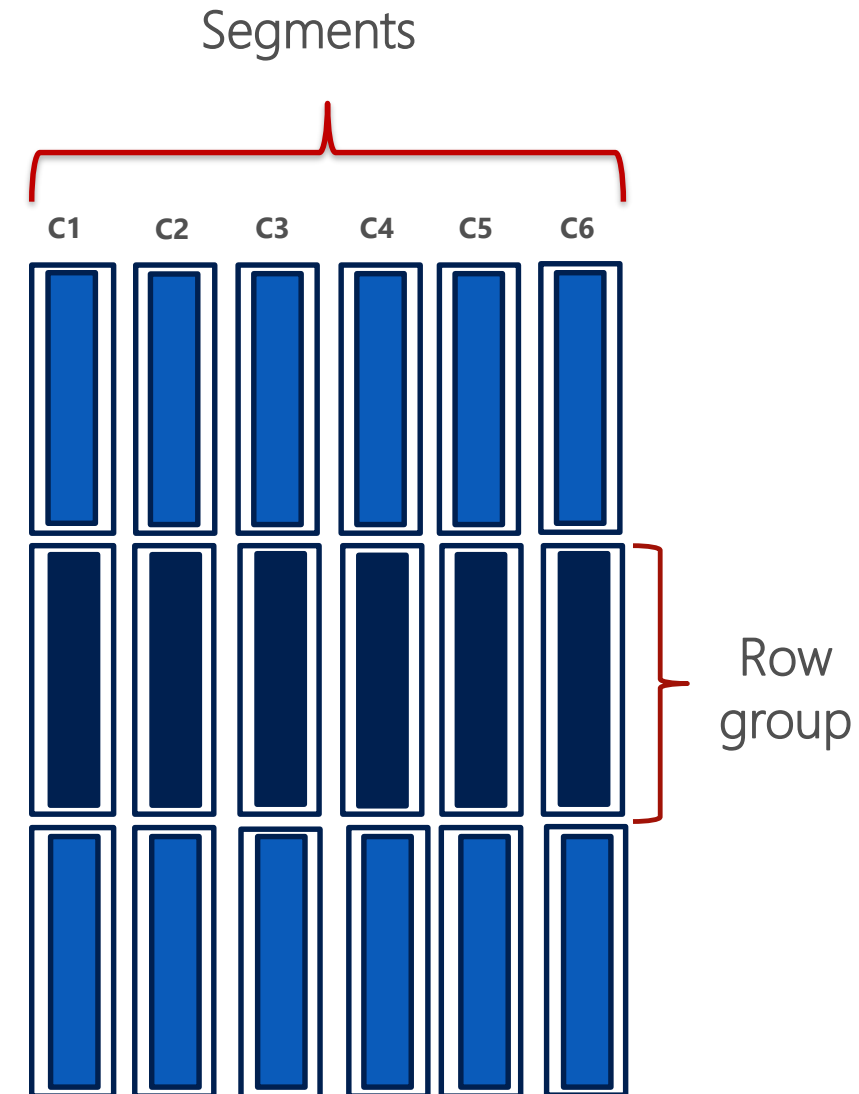
Segments are compressed

Each segment stored in a separate LOB

Segment is unit of transfer between disk and memory

Row group

Segments for the same set of rows comprise a row group



Columnstore Index Processing Example

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

Horizontally Partition - Row Groups

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00

~ 1M
rows

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

Vertical Partition - Segments

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00
OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

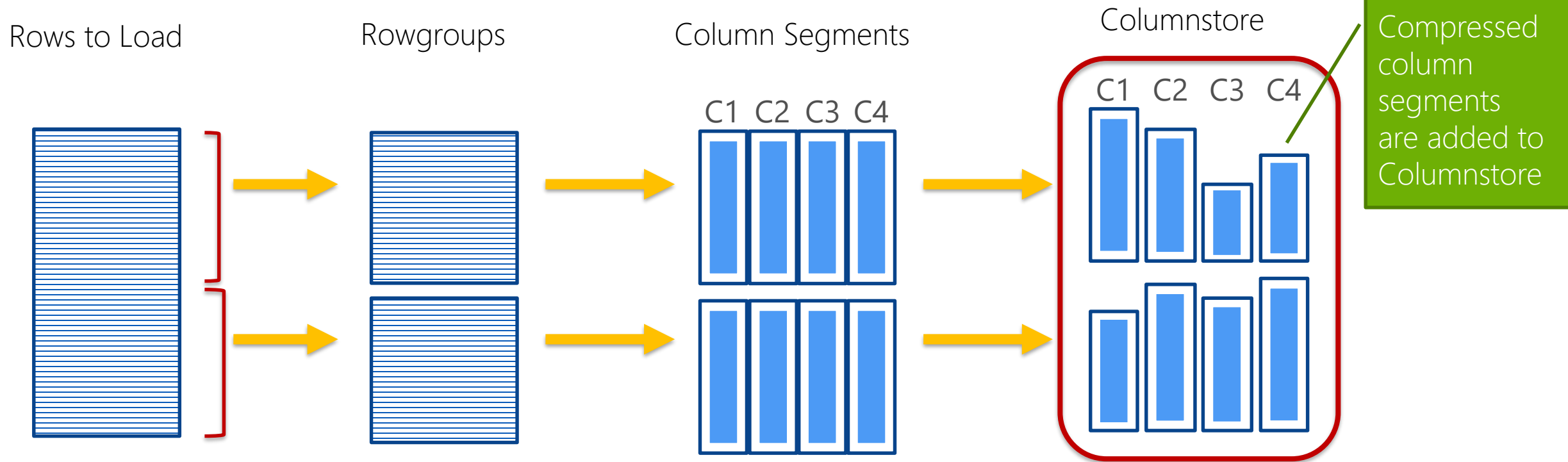
Compress Each Segment*

Some Compress More than Others

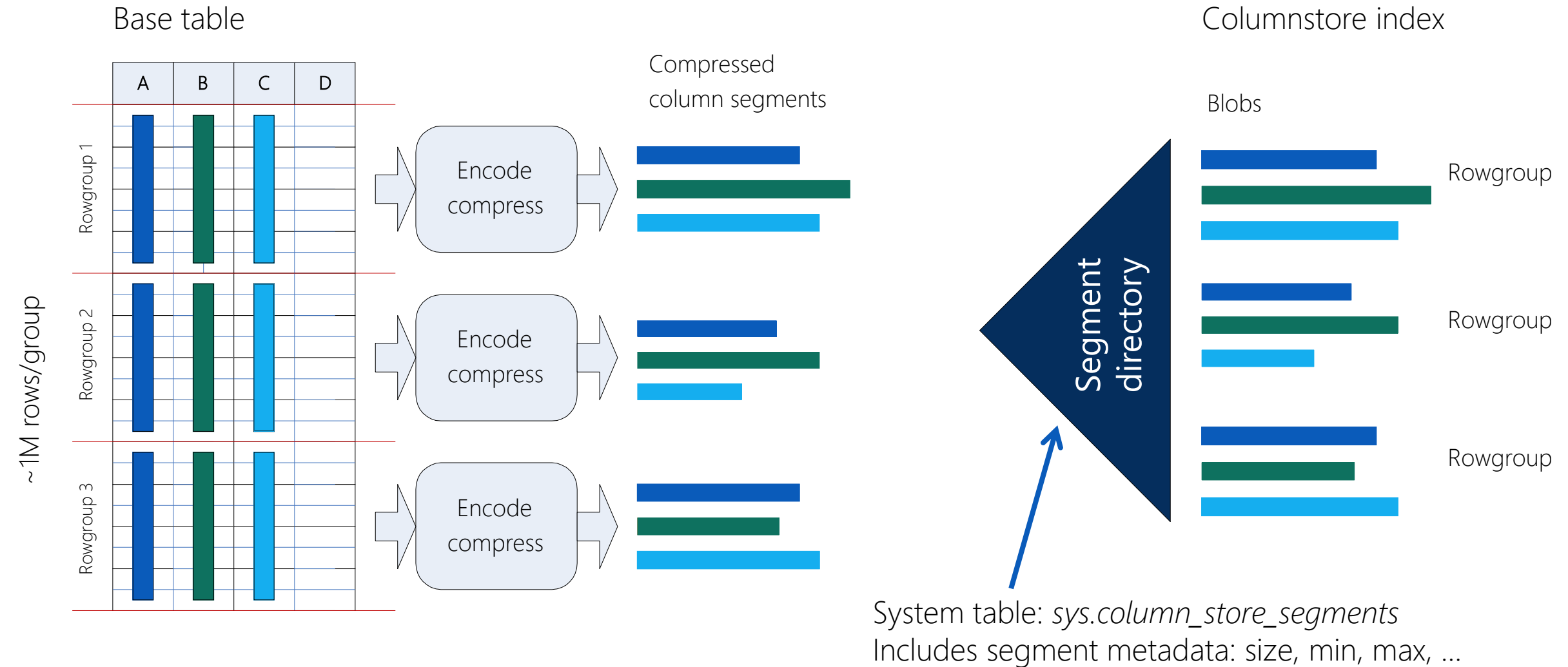
OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101108	106	05	3	4	20.00
20101108	106	02	1	5	25.00
OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101109	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	1	5	25.00
20101109	103	01	1	1	17.00

*Encoding and reordering not shown

Concepts Coming Together: Loading Data into a Nonclustered Columnstore Index



Index Creation and Storage



Syntax

```
CREATE CLUSTERED COLUMNSTORE INDEX CL_Simple  
ON SIMPLETABLE  
WITH (MAXDOP = 0)  
ON PRIMARY;
```

```
CREATE COLUMNSTORE INDEX NCI_Simple  
ON SIMPLETABLE (  
    SimpleID,  
    SimpleAddressID,  
    SimpleStateID,  
    Amt  
);
```

Have to specify columns
for nonclustered
columnstore index

```
CREATE CLUSTERED COLUMNSTORE INDEX CL_Simple  
ON SIMPLETABLE  
WITH (DROP_EXISTING = ON)  
ON PRIMARY;
```

Required if there is an
existing clustered index /
columnstore index

Limitations and Restrictions

Combination with non-clustered indexes

A table with a clustered columnstore index cannot have any type of nonclustered index

Constraints

A table with a clustered columnstore index cannot have unique constraints, primary key constraints, or foreign key constraints

View

Cannot be created on a view or indexed view

Keywords

Cannot be created by using the INCLUDE, ASC and DESC keyword

Unsupported Data Types

Following data types are not supported

- ntext, text, and image
- varchar(max) and nvarchar(max)
- rowversion (and timestamp)
- sql_variant
- CLR types (hierarchyid and spatial types)
- xml

Updatable Columnstore Index

Table consists of column store and row store

DML (update, delete, insert) operations leverage delta store

INSERT Values

Always lands into delta store*

DELETE

Logical operation

Data physically remove after REBUILD operation is performed.

UPDATE

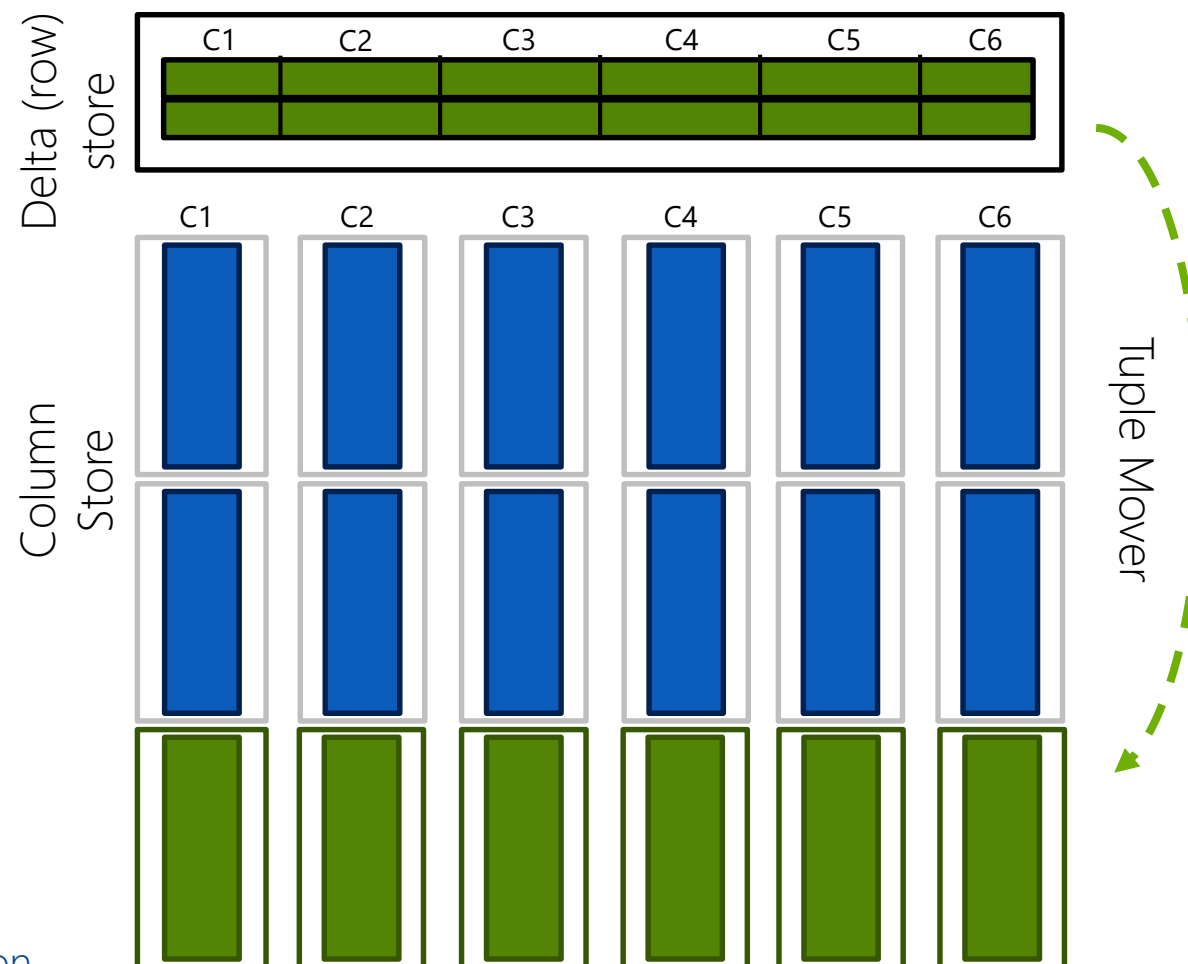
DELETE followed by INSERT.

BULK INSERT

if batch < 100k, inserts go into delta store, otherwise columnstore

SELECT

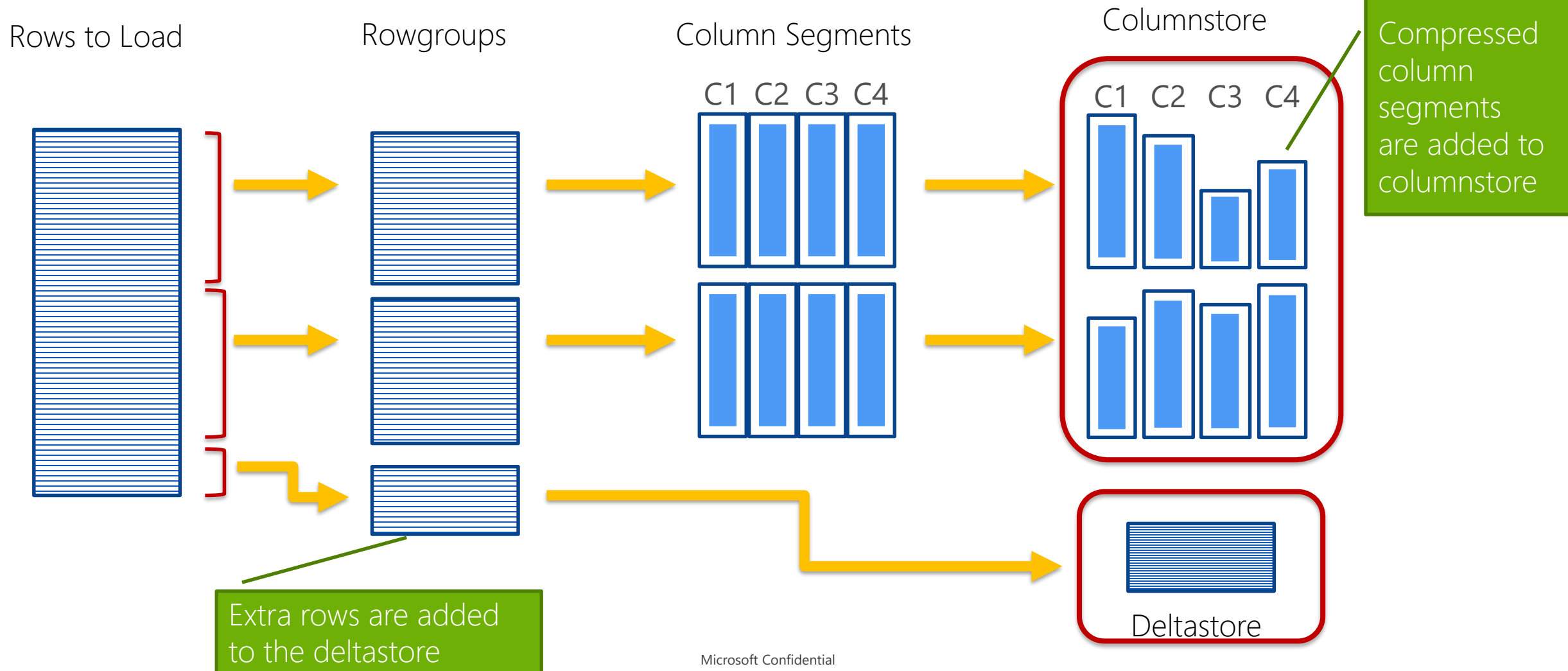
Unifies data from Column and Row stores - internal UNION operation.



“Tuple mover” converts data into columnar format once segment is full (1M of rows)

REORGANIZE statement forces Tuple Mover to start.

Concepts Coming Together: Loading Data into a Columnstore Index



Bulk Insert/Insert How It Works

BULK INSERT

Creates new columnstore row groups

INSERT

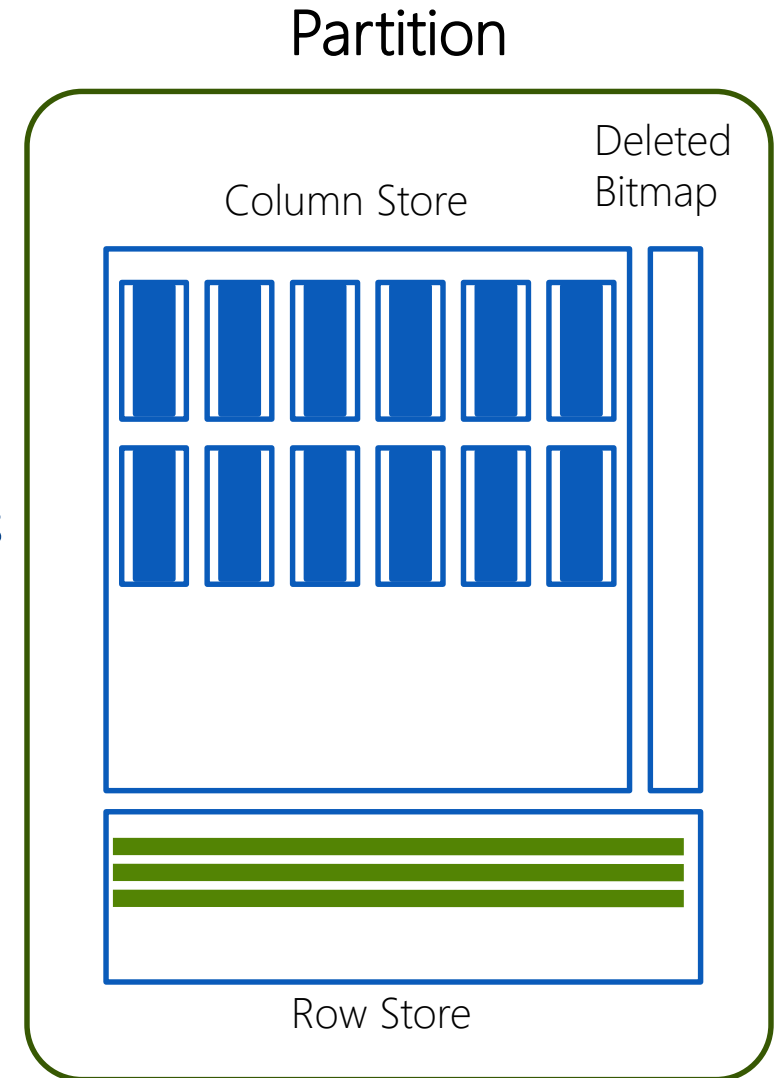
Rows are placed in the row store (heap)

When row store is big enough, a new columnstore row group is created

MINIMAL LOGGING

Applicable for delta store in Simple or Bulk-Logged Recovery model

- TABLOCK HINT
- Trace Flag 610



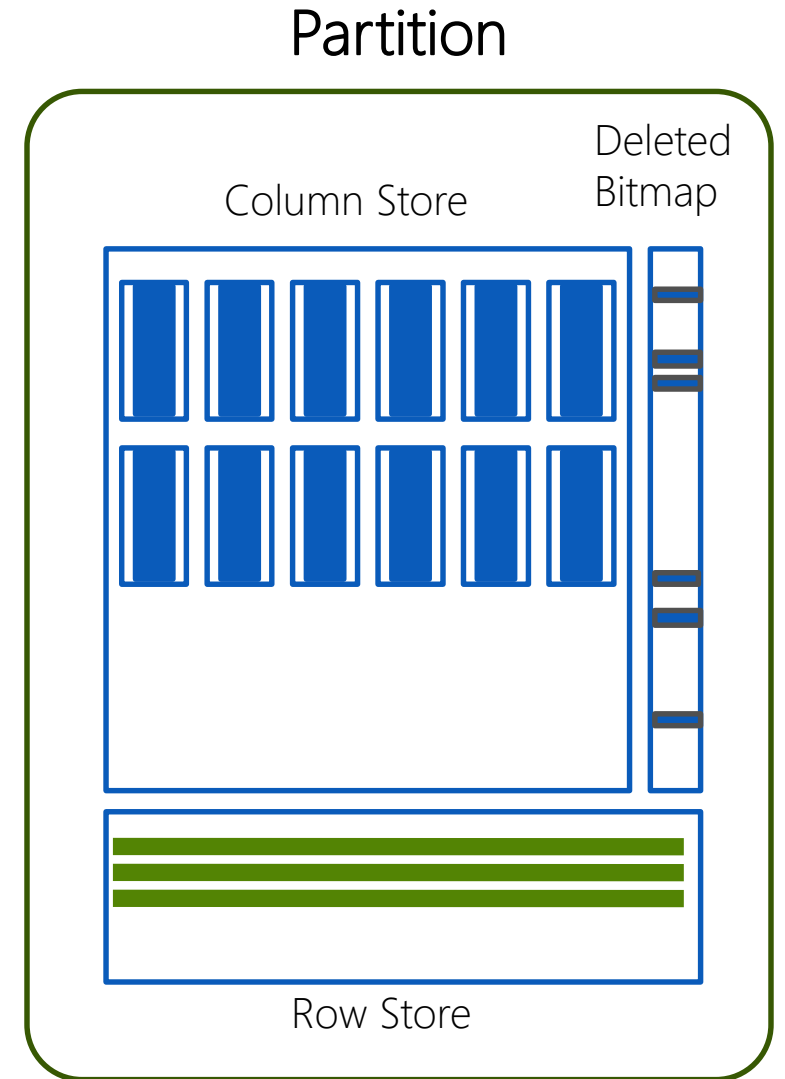
Delete/Update How It Works

DELETE

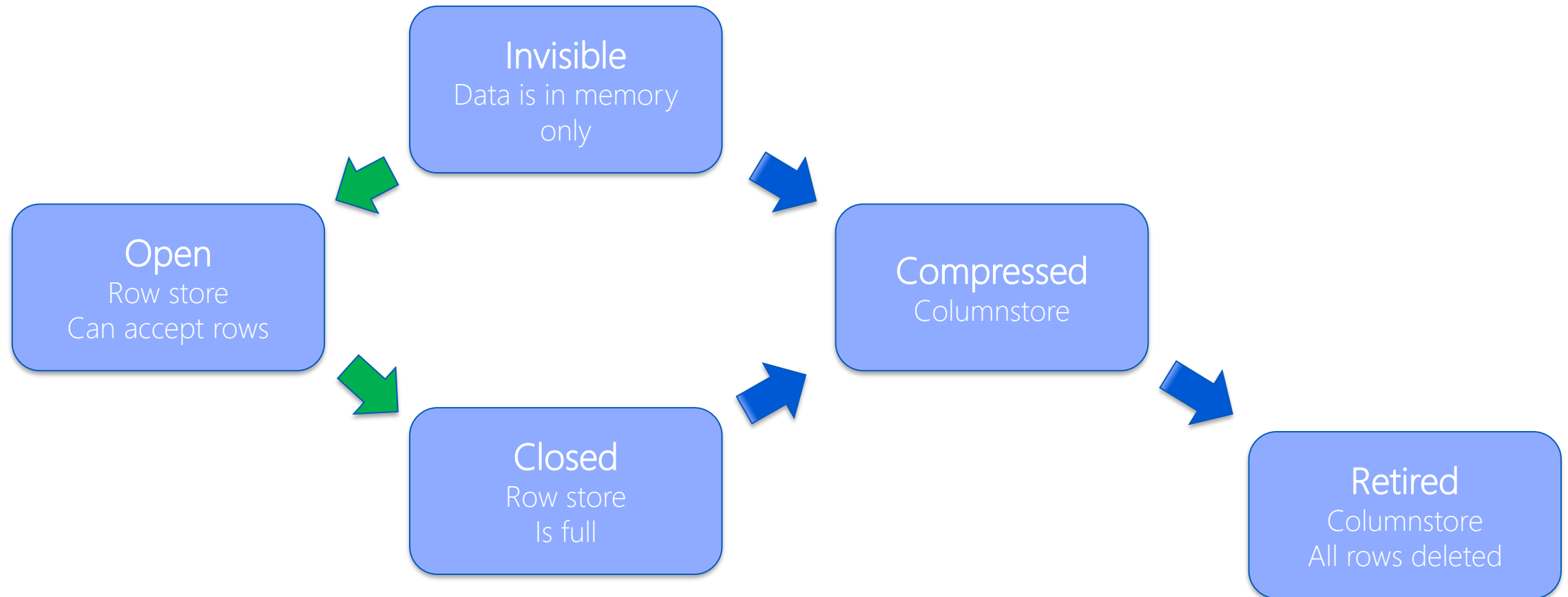
Rows are marked in the deleted bitmap

UPDATE

Delete plus insert



RowGroup State Changes



RowGroup DMV

```
SELECT *  
FROM sys.column_store_row_groups
```

Row store or deltastore
can accept rows

Columnstore

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
597577167	1	1	3	72057594044547072	1	OPEN	849	NULL	NULL
597577167	1	1	2	NULL	3	COMPRESSED	1048576	0	2814152
597577167	1	1	1	72057594044481536	2	CLOSED	1048576	NULL	NULL

Each row group has its own
deltastore

Closed (Full)
Waiting to be compressed

*

RETIRED – All rows deleted

INVISIBLE – Data in memory only

Bulk Insert Optimizations

Threshold for Tuple Mover is now 102,400 rows

- < 102,400 Rows inserted into delta store
- \geq 102,400 rows directly into columnstore
- If greater than 1,048,576 then rowgroup size will be limited to 1,048,576

Less than full columnstore row groups created by bulk insert will not be consolidated

- Batches of 90K row inserts: you eventually get large segments
- Batches of 120K row inserts: you get many 120K segments, and performance may not be as optimal long term
- Index rebuild will fix this by defragging the index, but that is resource intensive
- Physical order of data file determines how segments are created

Bulk Insert Optimizations

Bulk Insert < 102,400 Rows

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
757577737	1	1	0	72057594048544768	1	OPEN	50000	NULL	NULL

Bulk Inserted – 105K Rows ($\geq 102,400$ Rows)

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
773577794	1	1	3	NULL	3	COMPRESSED	105000	0	12312424
773577794	1	1	2	NULL	3	COMPRESSED	105000	0	12312424
773577794	1	1	1	72057594048806912	1	OPEN	1	NULL	NULL
773577794	1	1	0	NULL	3	COMPRESSED	105000	0	12312424

ALTER INDEX... REBUILD

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
773577794	1	1	0	NULL	3	COMPRESSED	315001	0	14825040

Tuple Mover

Runs every 5 minutes by default

- When row store reaches 1,048,576 rows convert to a columnstore
- De-allocates row groups where all rows are deleted

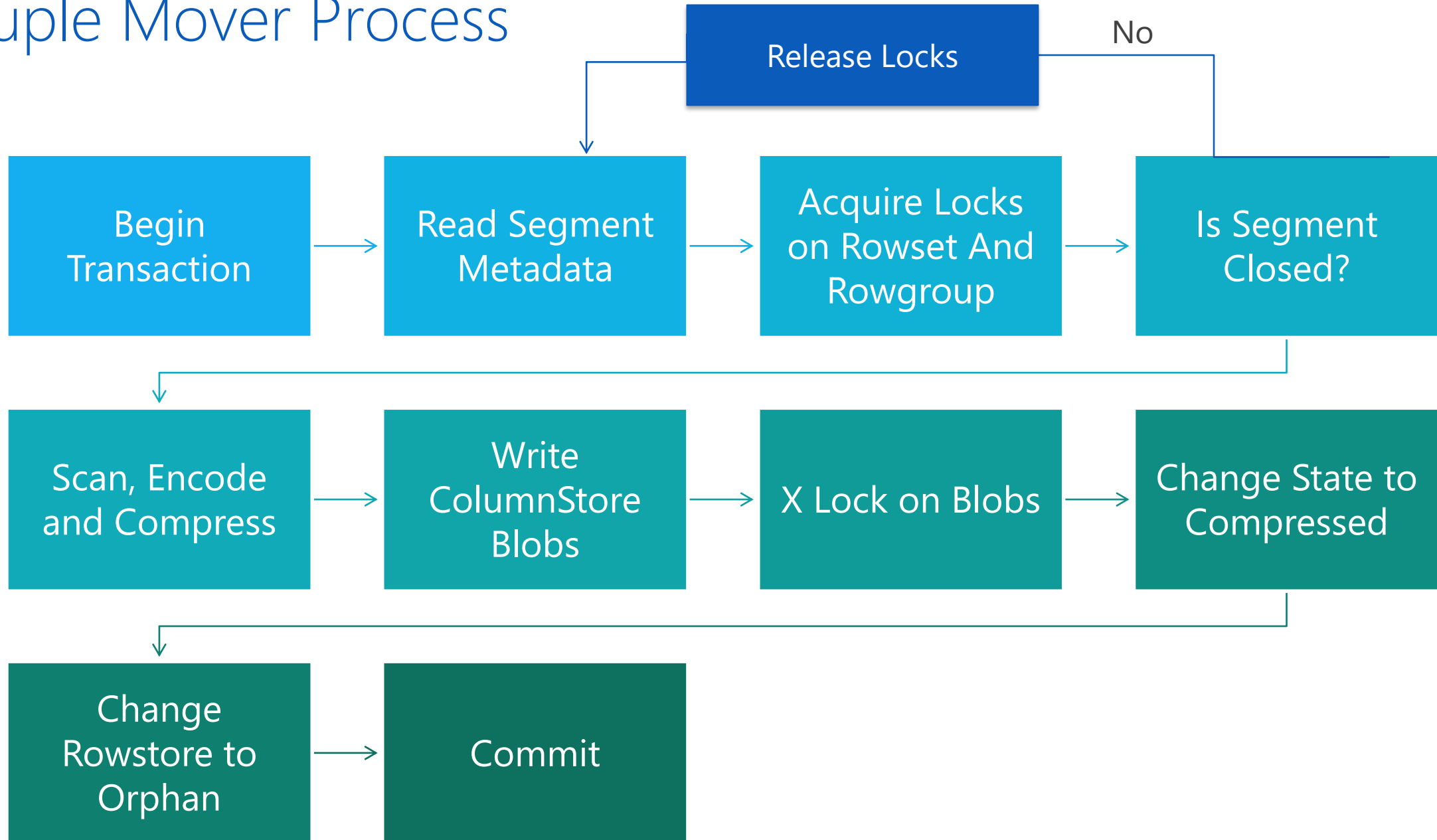
Start manually

- ALTER INDEX ...REORGANIZE

Extended events

- columnstore_tuple_mover_begin_compress
- columnstore_tuple_mover_end_compress

Tuple Mover Process



Tuple Mover Control

Tuple Mover does consume resources

Trace flag to disable Tuple Mover (634) as an edge case

When disabled, has to be manually invoked with:

Alter Index (...) Reorganize/Rebuild

If disabled and not manually invoked:

- Can cause performance issues when querying data
- Can end up with multiple rowstores (deltastore) which won't be compressed

Index Maintenance Operations

Index rebuild:

Re-creates clustered columnstore index completely

```
ALTER TABLE ... REBUILD
```

```
ALTER INDEX ... REBUILD
```

```
CREATE CLUSTERED COLUMNSTORE INDEX ... WITH (DROP_EXISTING = ON)
```

Reorganize:

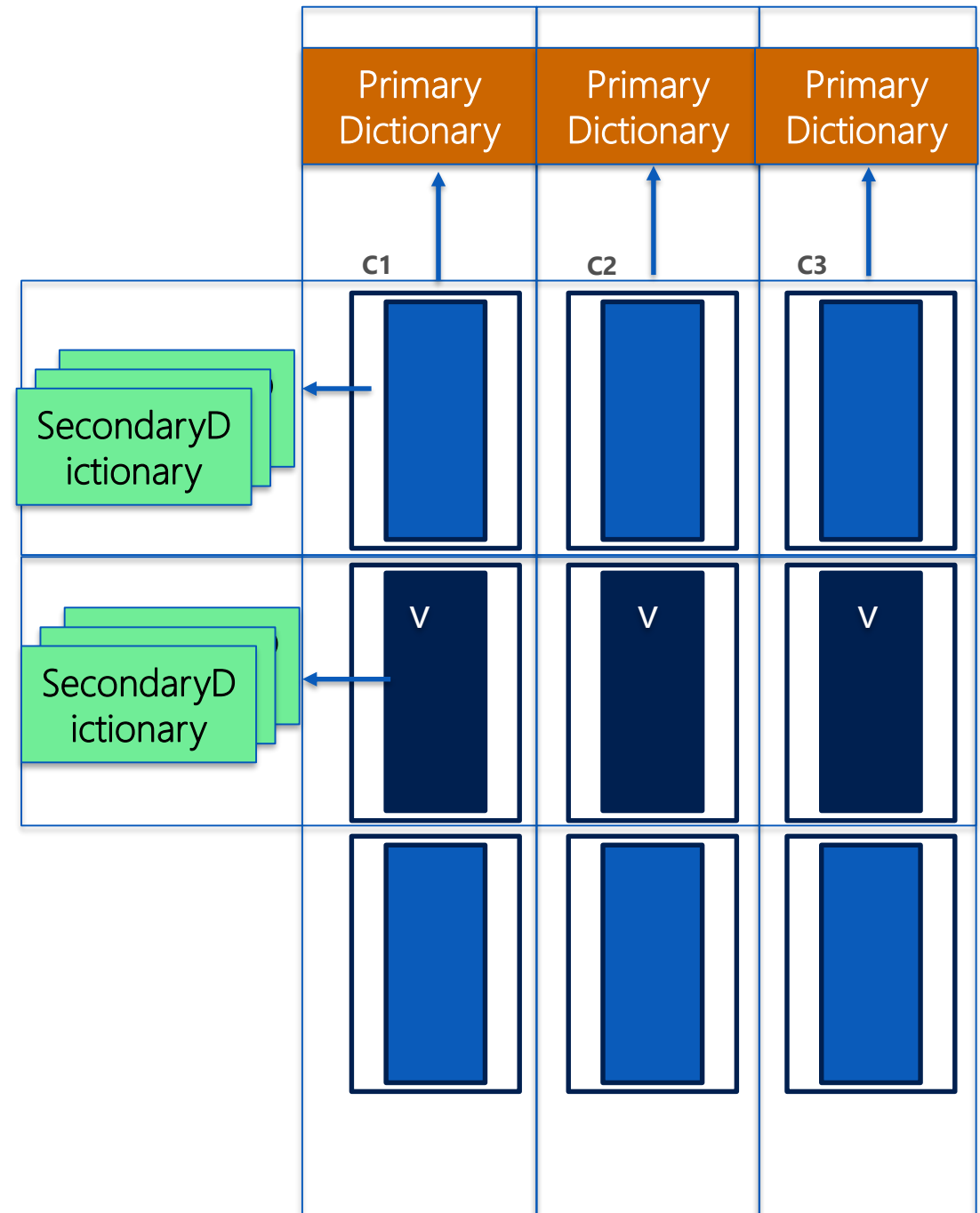
Forces delta store operations only

```
ALTER INDEX ... REORGANIZE // compresses closed row groups
```

```
... REORGANIZE WITH (COMPRESS_ALL_ROW_GROUPS = ON) // compresses all row groups
```

Dictionaries

- Used to encode data types like strings
- Values in segments are bookmarks to dictionary
- Dictionaries not shared across columns
- Primary dictionary is one per partition
- Secondary 1 or more per segment
- Space of the table includes storage space for dictionaries



Enhanced Compression

Table compression options:

DATA_COMPRESSION = { NONE | ROW | PAGE | COLUMNSTORE | COLUMNSTORE_ARCHIVE }

1. COLUMNSTORE Compression

- Default compression when creating a table with a clustered columnstore index
- Typical customer workloads gets 5-7x compression ratios

TPCH	3.1X
TPCDS	2.8X
Customer 1	3.9X
Customer 2	4.3X

** compression measured against raw data file

Enhanced Compression

Table compression options:

DATA_COMPRESSION = { NONE | ROW | PAGE | COLUMNSTORE | COLUMNSTORE_ARCHIVE }

ARCHIVAL compression

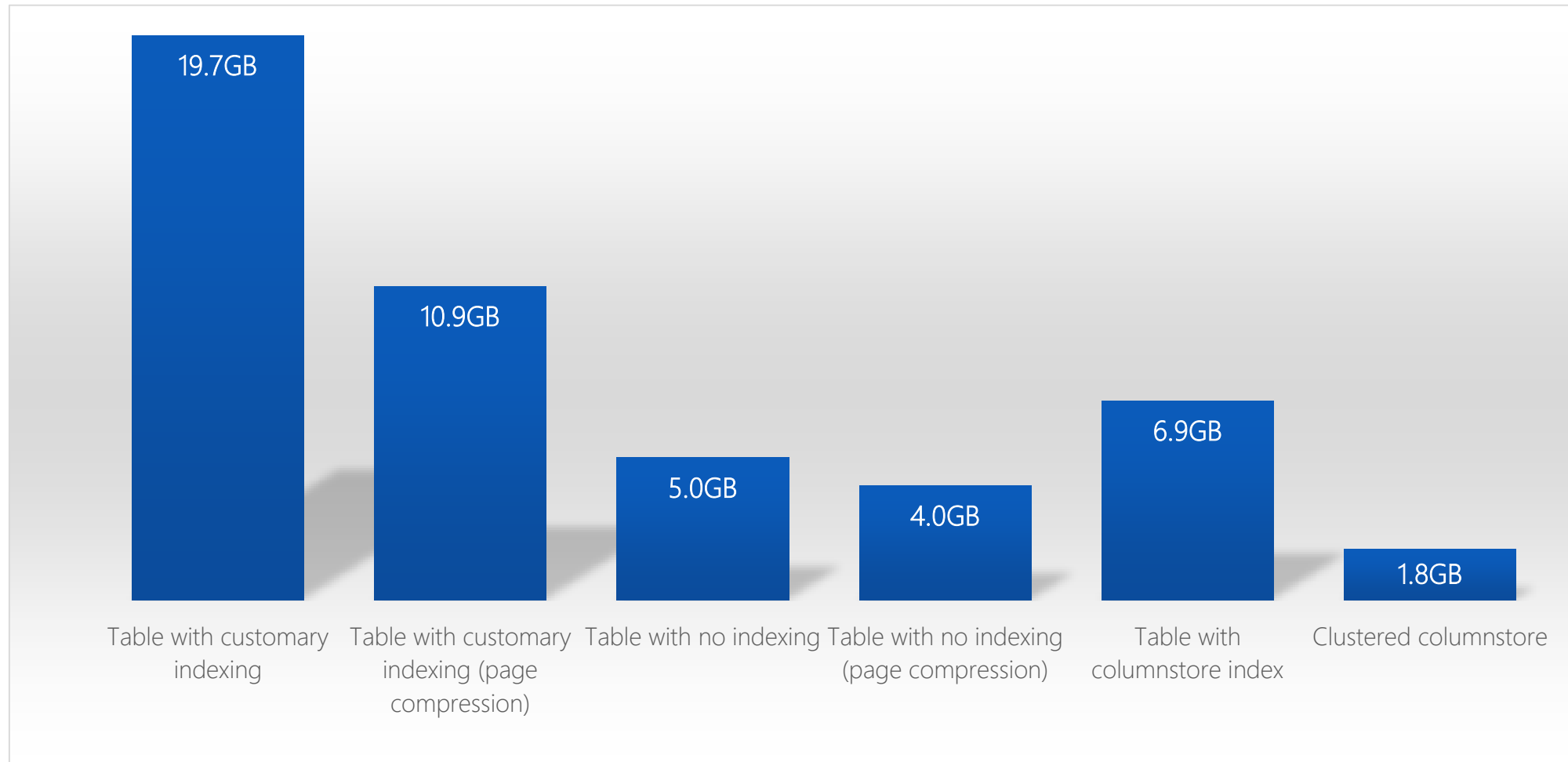
- Enables additional **30%** compression for whole table and/or chosen partitions
- Going back and forth between columnstore and columnstore_archive compressions
- **sys.partitions** exposes compression info (3 – columnstore, 4 – columnstore_archive)

```
ALTER TABLE FactResellerSalesPart_Big  
REBUILD PARTITION = 1  
WITH (DATA_COMPRESSION = COLUMNSTORE_ARCHIVE);
```

```
ALTER TABLE FactResellerSalesPart_Big  
REBUILD PARTITION = ALL  
WITH (DATA_COMPRESSION = COLUMNSTORE_ARCHIVE);
```

Comparing Space Savings of Columnstore Format

101 million row table + index space



Compression Ratios

The compression ratios

- Achieved with and without archival compression for several real data sets

The further reduction of archival compression

- Ranging from 37% to 66%

Database Name	Raw data size (GB)	Compression ratio		
		Archival compression?		GZIP
		No	Yes	
EDW	95.4	5.84	9.33	4.85
Sim	41.3	2.2	3.65	3.08
Telco	47.1	3.0	5.27	5.1
SQM	1.3	5.41	10.37	8.07
MS Sales	14.7	6.92	16.11	11.93
Hospitality	1.0	23.8	70.4	43.3

Source: <http://research.microsoft.com/apps/pubs/default.aspx?id=193599>

ColumnStore Cache Store

Memory Brokers enable larger memory consumers to adjust their usage based on others and respond to pressure

Broker for cache stores is MEMORYBROKER_FOR_CACHE

New Cachestore for Columnstore CACHESTORE_COLUMNSTOREOBJECTPOOL

- Execution Memory grants still obtained from the Query Execution Memory
- LOB Pages still use Buffer Pool

MEMORYBROKER_FOR_CACHE
Cache Stores

MEMORYBROKER_FOR_STEAL
Optimizer Memory , Default
for Clerks

MEMORYBROKER_FOR_RESERVE
Query Execution Memory
(Hashes and Sorts)

Caches and Clerks

sys.dm_os_memory_clerks

type	name	memory_node_id	pages_kb
CACHESTORE_COLUMNSTOREOBJECTPOOL	Column store object pool	0	8104

sys.dm_os_memory_brokers

pool_id	memory_broker_type	allocations_kb	allocations_kb_per_sec	predicted_allocations_kb	target_allocations_kb
2	MEMORYBROKER_FOR_CACHE	603928	0	603928	2626560

Extended Events – large_cache_*

name	timestamp	database_name	size_in_pages	session_id	is_insertion	sql_text
large_cache_entry_value_change	2014-02-21 13:03:35.8110188	AdventureWorks...	357	53	NULL	select f.SalesTerr...
large_cache_caching_decision	2014-02-21 13:03:35.8110412	AdventureWorks...	357	53	True	select f.SalesTerr...

Extended Events – column_store_*

name	timestamp	database_name	object_type	column_id	session_id	sql_text
column_store_object_pool_miss	2014-02-21 13:14:45.4974213	AdventureWorks...	COLUMN_SEGM...	4294967295	53	select f.SalesTerr...
column_store_rowgroup_read_issued	2014-02-21 13:14:45.4974544	AdventureWorks...	NULL	NULL	53	select f.SalesTerr...
column_store_object_pool_hit	2014-02-21 13:14:45.4981789	AdventureWorks...	COLUMN_SEGM...	3	53	select f.SalesTerr...

Building ColumnStore Indexes

Performance

- Memory resource intensive
- Memory requirement related to number of columns, data, DOP (Degree of Parallelism) #
- Creation can take 1.5 times longer than B-tree

SQL 2012

- Memory grant request in MB = $[(4.2 * \text{Number of columns in the CS index}) + 68] * \text{DOP} + (\text{Number of string cols} * 34)$
- Max memory grant is 25% of workspace memory by default

SQL 2014

- Total = (Per-thread consumption) * DOP
- Per-thread consumption is per-thread usage memory needed to build 1 segment

SQL Server 2012 Columnstore Index Build Characteristics

Number of rows per segment

- Each segment (except the last one) has a fixed number of rows: ~ 1 million

Memory size

- All columns included in the index have to be in memory to be processed

Required memory for string columns

- Memory used by string column depends not only on size but also on uniqueness of string

Memory size estimation

- SQL Server allowed the columnstore index build to exceed its initial memory estimate at runtime



Shortage of memory during build
= Errors 701, 802
Fail the query if estimate for DOP = 1 is >25%

Changes in 2014 Index build

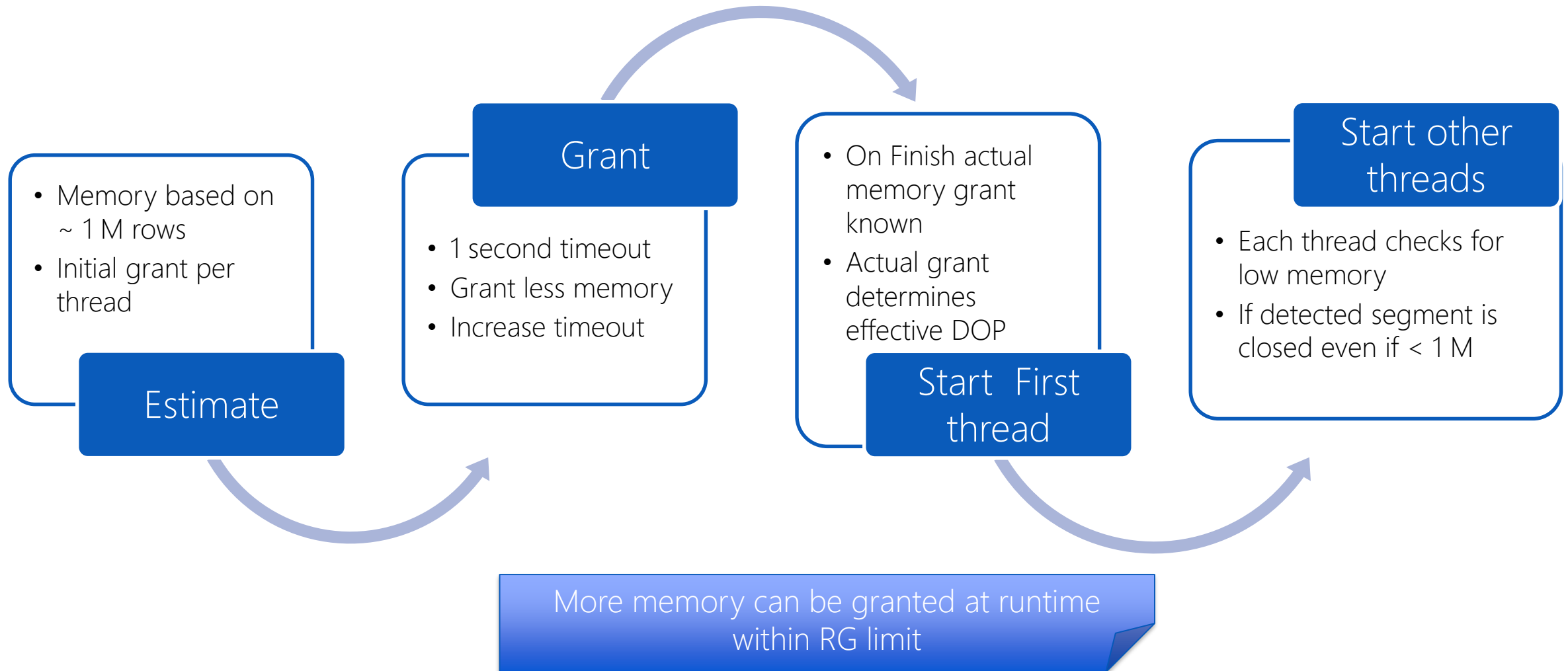
Streaming functionality for columnstore utilities (build, rebuild, load):

- Memory consumption adjusts under memory pressure (e.g. data load, index build/rebuild)
- Same memory grant and reservation process is being used by different processes (build/rebuild/load)

Available memory can affect columnstore segment quality

- **Ideal** segment size = 1M of rows
- Number of segments (columns in the table) drive memory requirements
- Attempts to create ideal segment by reserving “enough” memory
- Under memory pressure, DOP is being reduced first, **followed by segment size reduction**
- Resource Governor settings

Index Build Process



Why is my Degree of Parallelism (DOP) low?

DOP for query is low

- DOP for Query is set low either at Query or Server Level

Available memory low

- DOP is reduced
- Some threads suspended

Unit of parallelism is the segment

- Lots of segments, lots of parallelism

Effective DOP (# of threads running concurrently)

- All threads not started at the same time, can go up or down
- For example, DOP=8 columnstore index build can have 1 to 8 threads running concurrently

What if Index Build Still Fails?

Increase memory, max server memory option

Add physical memory to the system

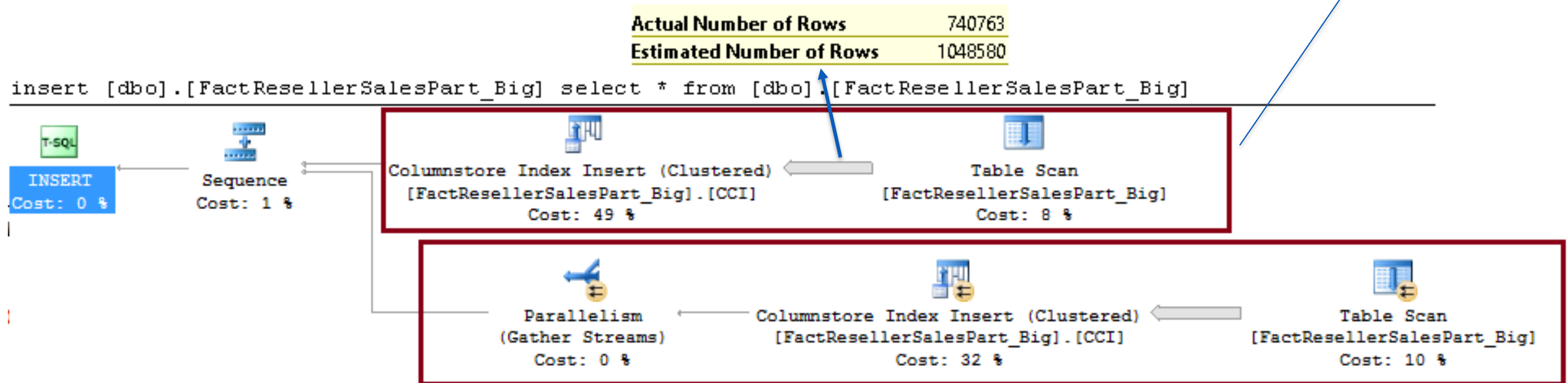
Reduce parallelism: (MAXDOP = 1);

Set REQUEST_MAX_MEMORY_GRANT_PERCENT to 50

```
ALTER WORKLOAD GROUP [DEFAULT] WITH (REQUEST_MAX_MEMORY_GRANT_PERCENT=X)  
ALTER RESOURCE GOVERNOR RECONFIGURE  
GO
```

Index Build Query Plans

Primary Dictionary built serially
Rows sampled – generally 1M or 1% of table



Segment building
Done in parallel if possible per DOP

Properties	
INSERT	
CompileTime	2
Degree of Parallelism	2
Effective Degree of Parallelism	1

Tracking Index Builds


Extended Events

Extended Event	Description
column_store_index_build_low_memory	Occurs when Storage Engine detects low memory condition and the rowgroup size is reduced.
column_store_index_build_throttle	Shows the statistics of columnstore index build parallelism throttling
column_store_index_build_process_segment	Shows the processing of every row group

name	timestamp	row_count	segment_count	max_effective_d...	effective_dop
column_store_index_build_process_segment	2014-02-21 17:02:...	NULL	NULL	NULL	NULL
column_store_index_build_throttle	2014-02-21 17:02:...	1397838	3	1	0
column_store_index_build_process_segment	2014-02-21 17:04:...	NULL	NULL	NULL	NULL
column_store_index_build_low_memory	2014-02-21 17:04:...	NULL	NULL	NULL	NULL

Isolation Levels

Isolation Level	Behavior
READ UNCOMMITTED	Read Uncommitted Data
READ COMMITTED	Read committed Data
REPEATABLE READ	Ensures repeatable reads
SERIALIZABLE	No Phantoms allowed
READ_COMMITTED_SNAPSHOT	Functionality not Supported (Inserts will block Readers) , but no errors thrown
SNAPSHOT	Not Supported



Minimum
Lock
granularity of
an entire
segment (row
group).

Concurrency

Locking is at the Rowgroup level

- Not Ideal for any OLTP workloads, designed for DW

Types of locks depend on

- Operation
- Isolation level
- Deltastore versus columnstore

request_session_id	request_mode	request_status	resource_type	resource_description	ObjectName	index_name	resource_associated_entity_id
52	X	GRANT	ROWGROUP	ROWGROUP: 11:1000000019c00002	t2	t2CCI	72057594064928768

sys.column_store_row_groups - row_group_id

sys.column_store_segments - hobt_id

Lock Types

Rowgroup Schema Stability

- Protects the lifetime of the rowgroup
- SCH-S by all transactions except creation and deletion of a rowgroup

Rowgroup Data Lock (RGDL)

- Access to data in the rowgroup
- S, IU, IX, U, X

Rowgroup Flush Lock (RGFL)

- Protects modifications to the rowgroup metadata
- Short lived

Rowgroup RS locks

- Normal rowstore access locking protocol
- At rowset/page/row level granularity

Partitioning Columnstore Indexed Table

Columnstore indexes are designed to support queries in very large data warehouse scenarios, where partitioning is common.

Partition 1

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00

Partition 2

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

Partitioning Information

sys.column_store_row_groups provides clustered columnstore index information on a per-segment basis

object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_description	total_rows	deleted_rows	size_in_bytes
901578250	1	1	2	NULL	3	COMPRESSED	1048576	0	4488728
901578250	1	1	0	72057594042712064	1	OPEN	451424	NULL	NULL
901578250	1	2	2	NULL	3	COMPRESSED	1048576	0	4536018
901578250	1	2	1	72057594042908672	1	OPEN	451424	NULL	NULL
901578250	1	3	2	NULL	3	COMPRESSED	1048576	0	4532820
901578250	1	3	1	72057594043039744	1	OPEN	451424	NULL	NULL
901578250	1	4	0	72057594043105280	1	OPEN	500000	NULL	NULL

Looking in the *total_rows* and *deleted_rows* column, determine which row groups have a high percentage of deleted rows and should be rebuilt