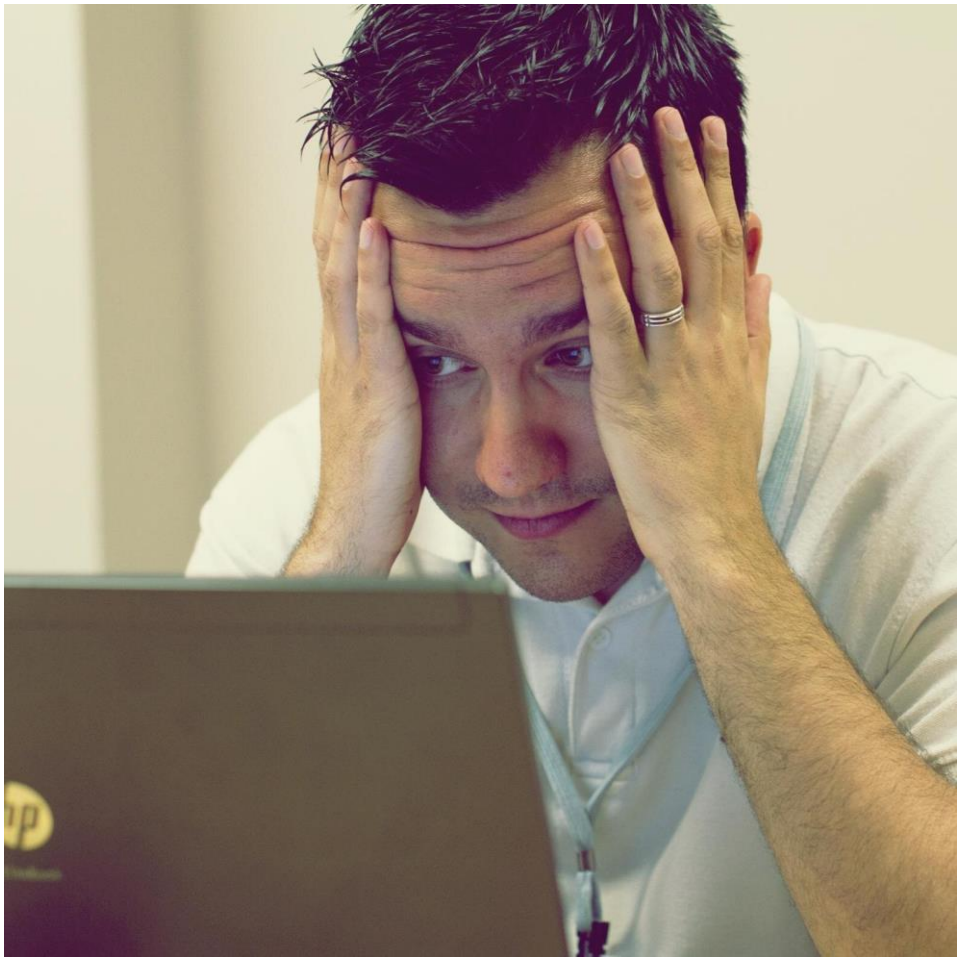
Clustered Columnstore – Deep Dive

Niko Neugebauer

Barcelona, October 25th 2014

Have you seen this guy ?



Our Main Sponsors:

PERSONA CIÈNCIA EMPRESA

Universitat Ramon Llull

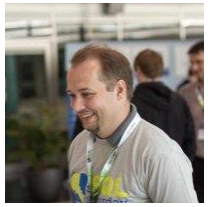
Microsoft

ApexSQL

SolidQ

ATTUNITY

Niko Neugebauer



Microsoft Data Platform Professional

OH22 (http://www.oh22.net)

15+ years in IT

SQL Server MVP

Founder of 3 Portuguese PASS Chapters

Blog: http://www.nikoport.com

Twitter: @NikoNeugebauer

LinkedIn: http://pt.linkedin.com/in/webcaravela

Email: info@webcaravela.com

So this is a supposedly Deep Dive ☺

- My assumptions:
  - You have heard about Columnstore Indexes
  - You understand the difference between RowStore vs Columnstore
  - You know about Dictionary existance in Columnstore Indexes
  - You know how locking & blocking works (at least understand the S, SI, IX, X locks)
  - You have used **DBCC Page** functionality ☺
  - You are crazy enough to believe that this topic could be expanded into this kind of level ☺

Todays Plan

- Intro (About Columnstore, Row Groups, Segments, Delta-Stores)
- Batch Mode
- Compresson phases
- Dictionaries
- Materialisation
- Meta-informations
- DBCC (Nope ☺)
- Locking & Blocking (Hopefully ☺)
- Bulk Load (Nope ☺)
- Tuple Mover (Nope ☺)

About Columnstore Indexes:

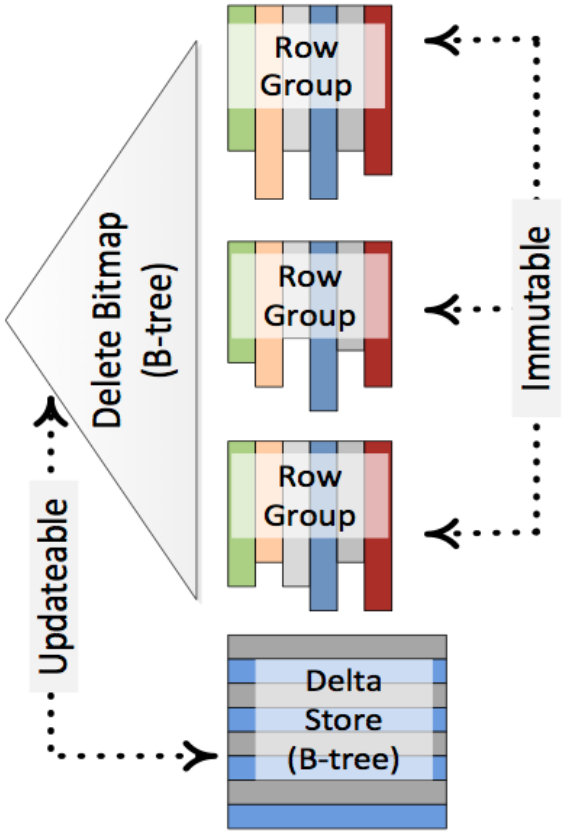
- Reading Fact tables
- Reading Big Dimension tables
- Very low-activity big OLTP tables, which are scanned & processed almost entirely

- Data Warehouses
- Decision Support Applications
- Business Intelligence Applications

Clustered Columnstore in SQL Server 2014

- Delta-Stores (open & close)
- Deleted Bitmap
- Delete & Update work as a

  DELETE + INSERT



BATCH MODE

Batch Mode

- New Model of Data Processing
- Query execution by using **GetNext** (), which delivers data to the CPU (In its turn it will go down the stack and get physical accecss to data. Every operator behaves this way, which makes **GetNext()** a virtuall function.
  For execution plans sometimes you will have 100s of this function invocation before you will get actual 1 row.
  For OLTP it might be a good idea, since we are working just with few rows, but If you are working with millions of rows (in BI or DW) you will make billions of such invocations.
- Entering Batch Mode, which actually invokes data for processing not 1 by 1 but in Batches of ~900 rows
- This might bring benefits in 10s & 100s times

Batch Mode

- In Batch Mode every operator down the stack have to play the same game, passing the same amount of data -> Row Mode can't interact with Batch Mode.
- **64 row** vs **900 rows**
  (Progammers, its like passing an **Array** vs **1 by 1 param**)
- Works exclusively for Columnstore Indexes
- Works exclusively for parallel plans, hence MAXDOP >= 2
- Think about it as if it would be a Factory processing vs *Manual Processing* (19th vs 18th Century)

Batch Mode is fragile

- Not every operator is implemented in Batch Mode.
- Examples of Row Mode operators: **Sort**, **Exchange**, **Inner LOOP**, **Merge**, ...
- Any disturbance in the force will make *Batch Execution Mode* to fall down into *Row Execution Mode*, for example lack of memory.
- SQL Server 2014 introduces so-called "Mixed Mode", where execution plan operators in Row Mode can co-exist with Batch Mode operators

Batch Mode Deep Dive

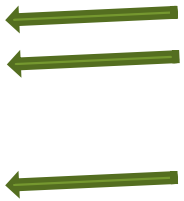
- Optimized for 64 bit values of the register
- Late materialization (working on compressed values)
- Batch Size is optimized to work in L2 Cache with idea of avoiding Cache Misses

Latency Cache

- L1 cache reference – 0.5 ns
- L2 cache reference – 7.0 ns (14 times slower)
- L3 cache reference – 28.0 ns (4 times slower)
- L3 cache reference (outside NUMA) – 42.0 ns (6 times slower)
- Main memory reference – 100ns (3 times slower)
- Read 1 MB sequentially from memory – 250.000 ns (5.000 times L1 Cache)

Batch Mode

Physical Operation	Columnstore Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	ColumnStore
Actual Number of Rows	12627608
Actual Number of Batches	14060
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.697651 (19%)
Estimated Subtree Cost	0.697651
Estimated CPU Cost	0.694526
Estimated Number of Executions	1
Number of Executions	2
Estimated Number of Rows	12627600
Estimated Row Size	9 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	5



SQL Server 2014 Batch Mode

- All execution improvements are done for Nonclustered & Clustered Columnstores
- Mixed Mode – Row & Batch mode can co-exist
- OUTER JOIN, UNION ALL, EXIST, IN, Scalar Aggregates, Distinct Aggregates – all work in Batch Mode
- Some **TempDB** operations for Columnstore Indexes are running in Batch mode. (TempDB Spill)

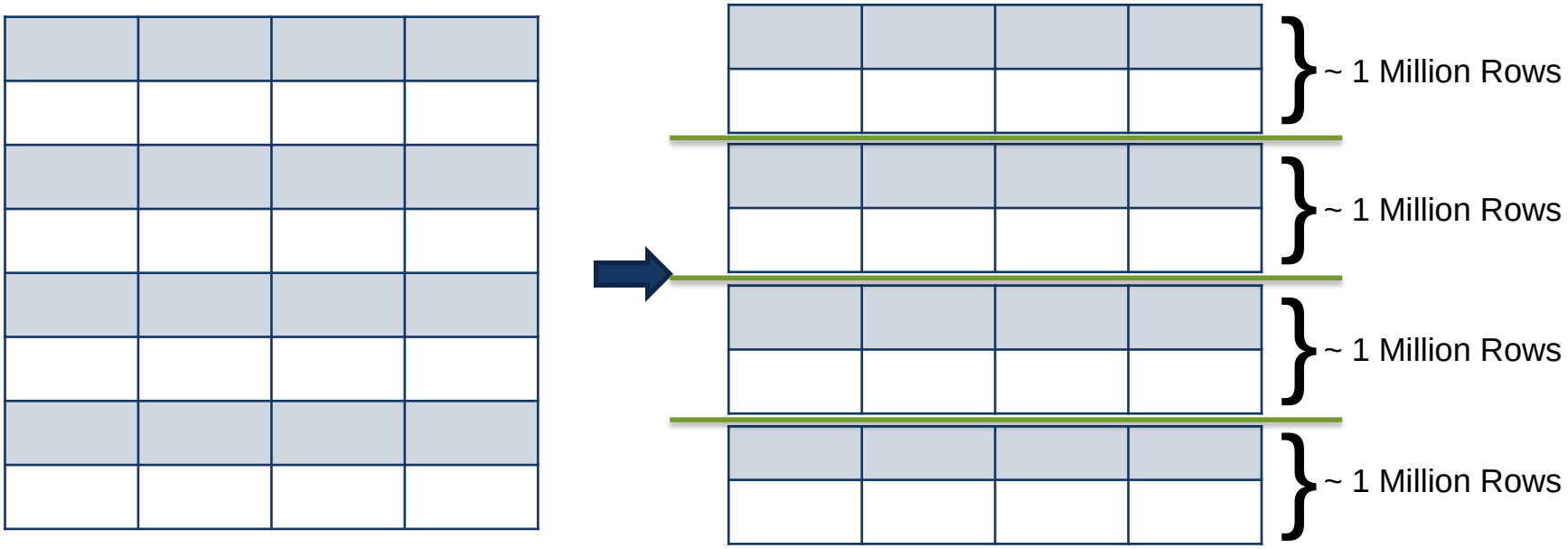
Demo, Demo, Demo

BATCH MODE

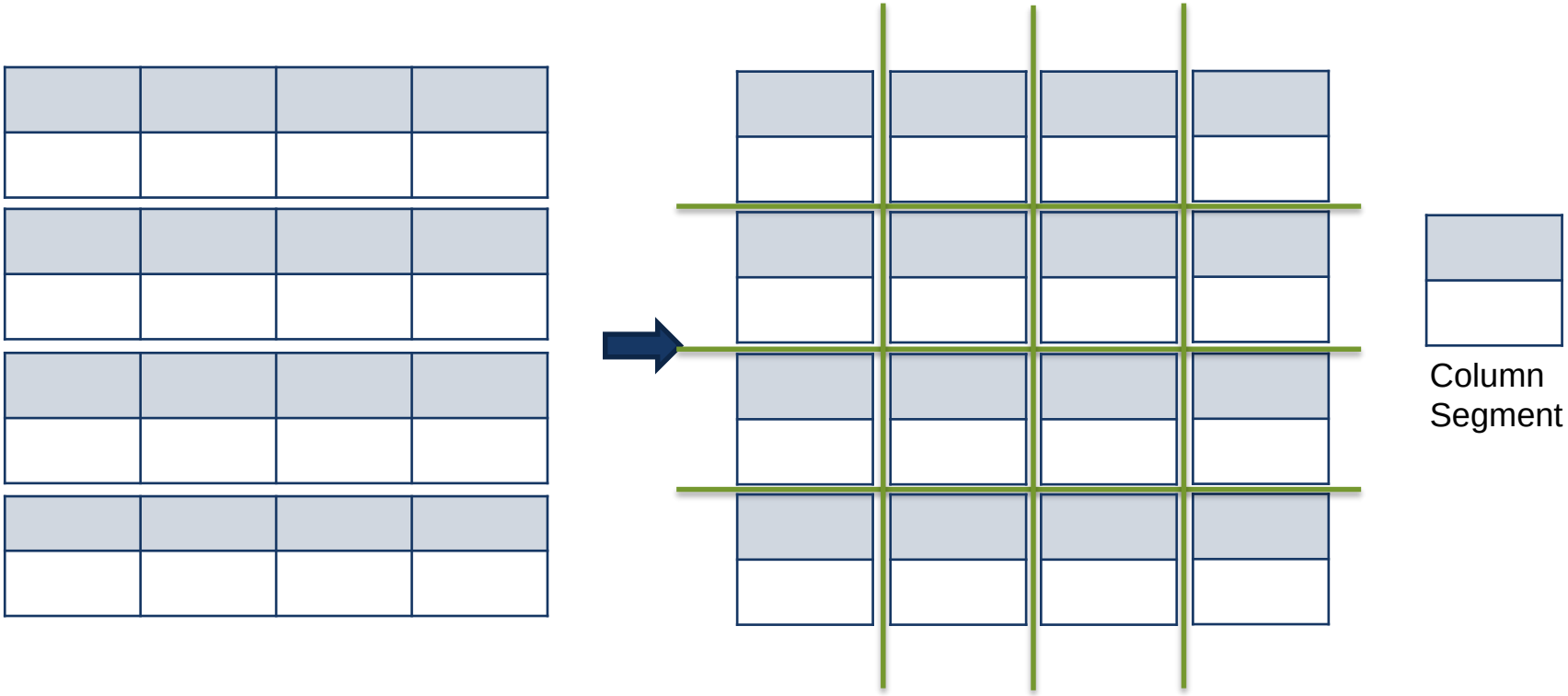
Basics phases of the Columnstore Indexes creation:

1. Row Groups separation
2. Segment creation
3. Compression

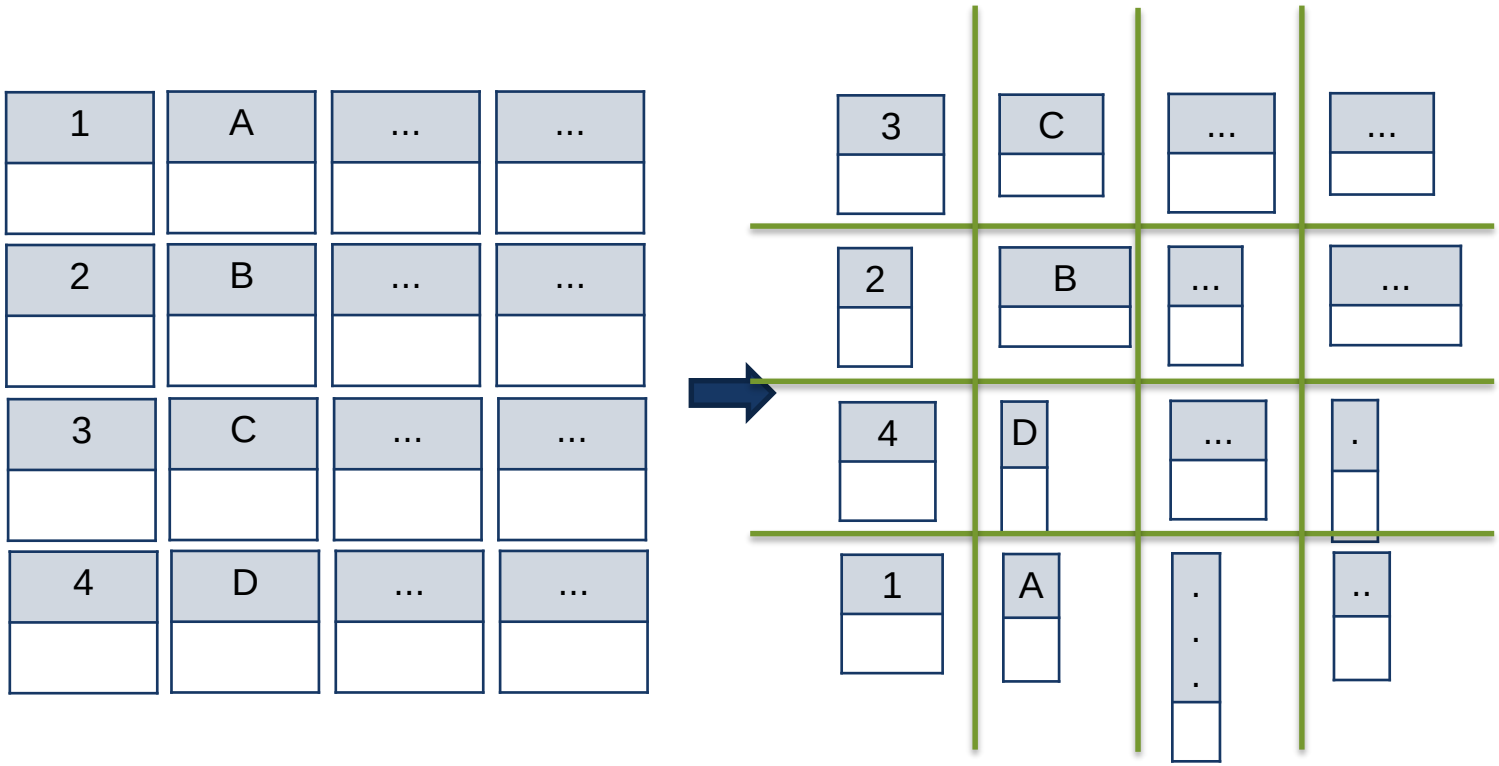
1. Row Groups creation



2. Segments separation



3. Compression (involves reordering, compression & LOB conversion)



Columnstore compression steps (when applicable)

- Value Scale
- Bit-Array
- Run-length compression
- Dictionary encoding
- Huffman encoding
- Binary compression

Value Scale

Amount
1023
1002
1007
1128
1096
1055
1200
1056

Amount
23
2
7
128
96
55
200
56

Base/Scale: 1000

Bit Array

Name
Mark
Andre
John
Mark
John
Andre
John
Mark

Mark	Andre	John
1	0	0
0	1	0
0	0	1
1	0	0
0	0	1
0	1	0
0	0	1
1	0	0

Run-Length Encoding (compress)

Name
Mark
Mark
John
Andre
Andre
Andre
Ricardo
Mark
Charlie
Mark
Charlie

Name
Mark:2
John:1
Andre:3
Ricardo:1
Mark-Charlie:2

Run-length compression, more complex scenario

Name	Last Name
Mark	Simpson
Mark	Donalds
John	Simpson
Andre	White
Andre	Donalds
Andre	Simpson
Ricardo	Simpson
Mark	Simpson
Charlie	Simpson
Mark	White
Charlie	Donalds

Name	Last Name
Mark	Simpson
Mark	Donalds
Mark	Simpson
Mark	White
John	Simpson
Andre	White
Andre	Donalds
Andre	Simpson
Ricardo	Simpson
Charlie	Simpson
Charlie	Donalds

Name	Last Name
Mark:4	Simpson:1
John:1	Donalds:1
Andre:3	Simpson:1
Ricardo:1	White:1
Charlie:2	Simpson:1
	White:1
	Donalds:1
	Simpson:3
	Donalds:1

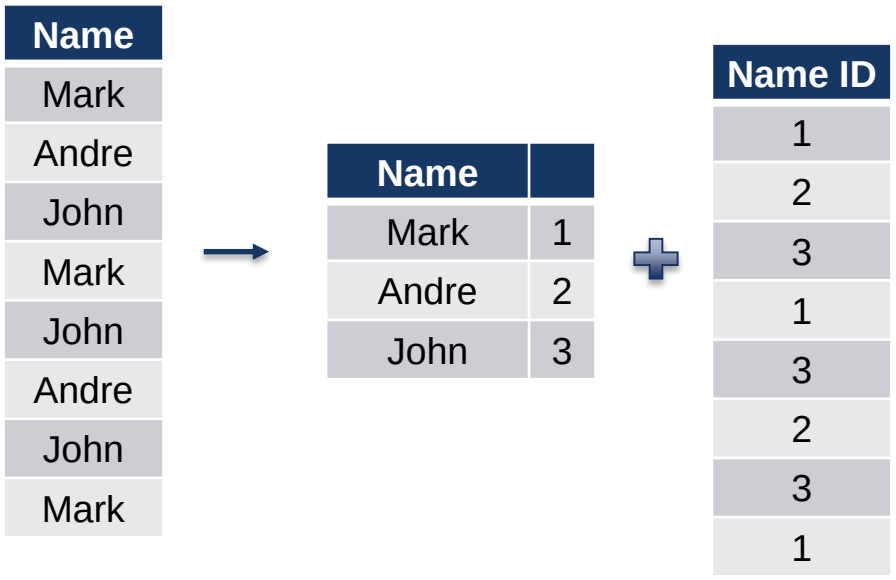
Run-length compression, more complex scenario, part 2

Name	Last Name
Mark	Simpson
Mark	Donalds
John	Simpson
Andre	White
Andre	Donalds
Andre	Simpson
Ricardo	Simpson
Mark	Simpson
Charlie	Simpson
Mark	White
Charlie	Donalds

Name	Last Name
Andre	Donalds
Charlie	Donalds
Mark	Donalds
Mark	Simpson
Mark	Simpson
Andre	Simpson
Ricardo	Simpson
John	Simpson
Charlie	Simpson
Andre	White
Mark	White

Name	Last Name
Andre:1	Donalds:3
Charlie:1	Simpson:6
Mark:3	White:3
Andre:1	
Ricarod:1	
John:1	
Charlie:1	
Andre:1	
Mark:1	

Dictionary enconding



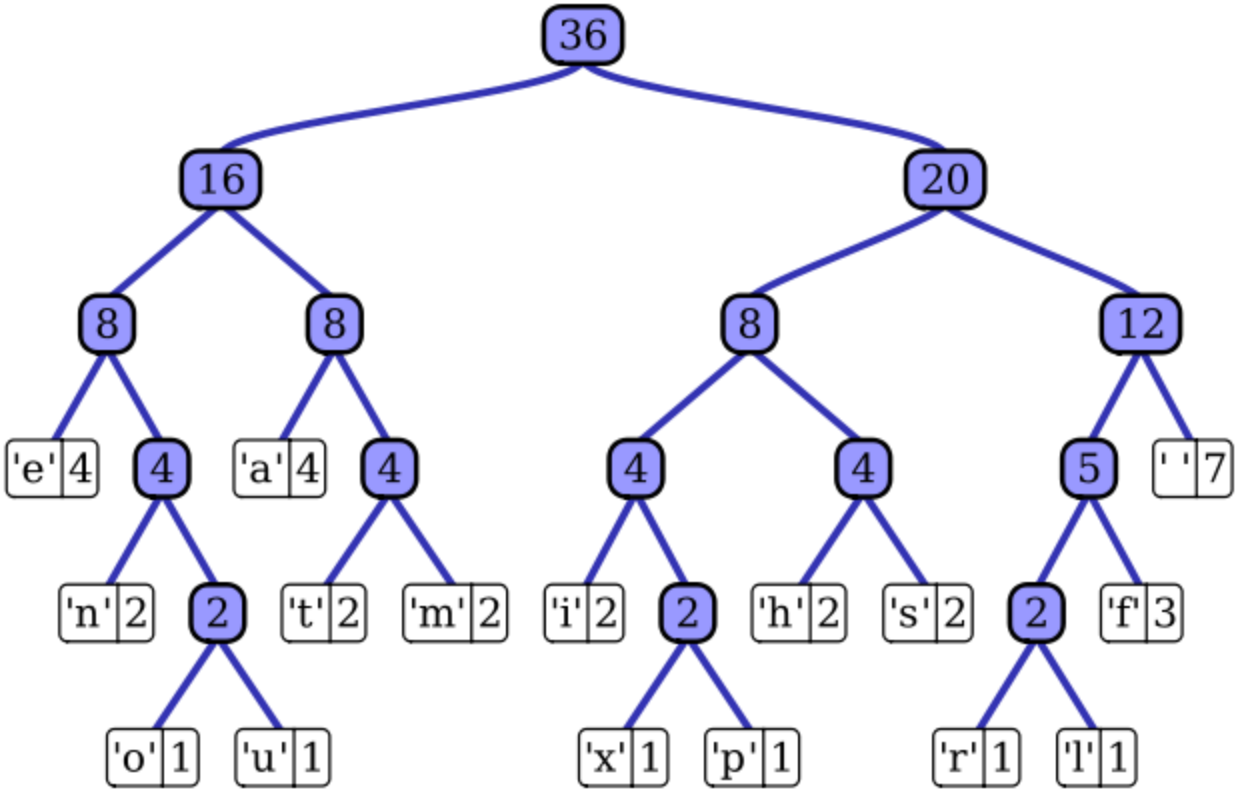
Huffman enconding (aka ASCII encoding)

- Fairly efficient ~ N log (N)
- Design a Huffman code in linear time if input probabilities (aka weights) are sorted.

Name	Last Name
Mark	Simpson
Mark	Donalds
Mark	Simpson
Mark	White
John	Simpson
Andre	White
Andre	Donalds
Andre	Simpson
Ricardo	Simpson
Charlie	Simpson
Charlie	Donalds

Name	Count	Code
Mark	4	001
Andre	3	010
Charlie	2	011
John	1	100
Ricardo	1	101

Huffman enconding tree (sample)



Binary Compression

- Super-secret **Vertipac** aka **xVelocity** compression turning data into LOBs. ☺
- LOBs are stored by using traditional storage mechanisms (8K pages & extents)

COLUMNSTORE ARCHIVE

Columnstore Archival Compression

- One more compression level
- Applied over the *xVelocity* compression
- It is a slight modification of **LZ77** (aka Zip)

Compression Recap:

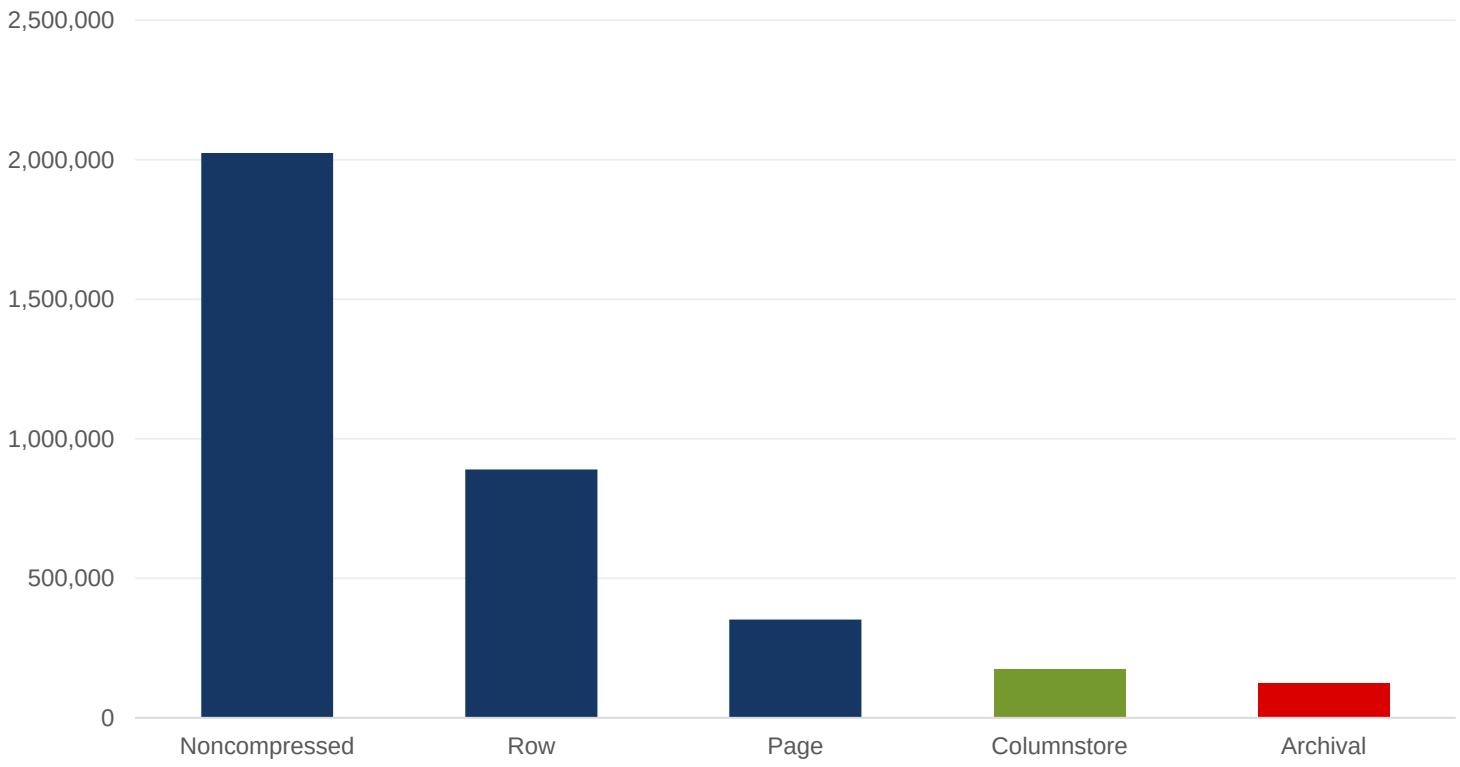
Determination of the best algorithm is the principal key for the success for the X-Velocity. This process includes data shuffling between segments and different methods of compression.

Every segment has different data, and so different algorithms with different success are being applied.

If you are seeing a lot of queries including a predicate on a certain column, then try creating a traditional clustered index on it (sorting) and then create a columnstore.

Every compression is supported on the partition level

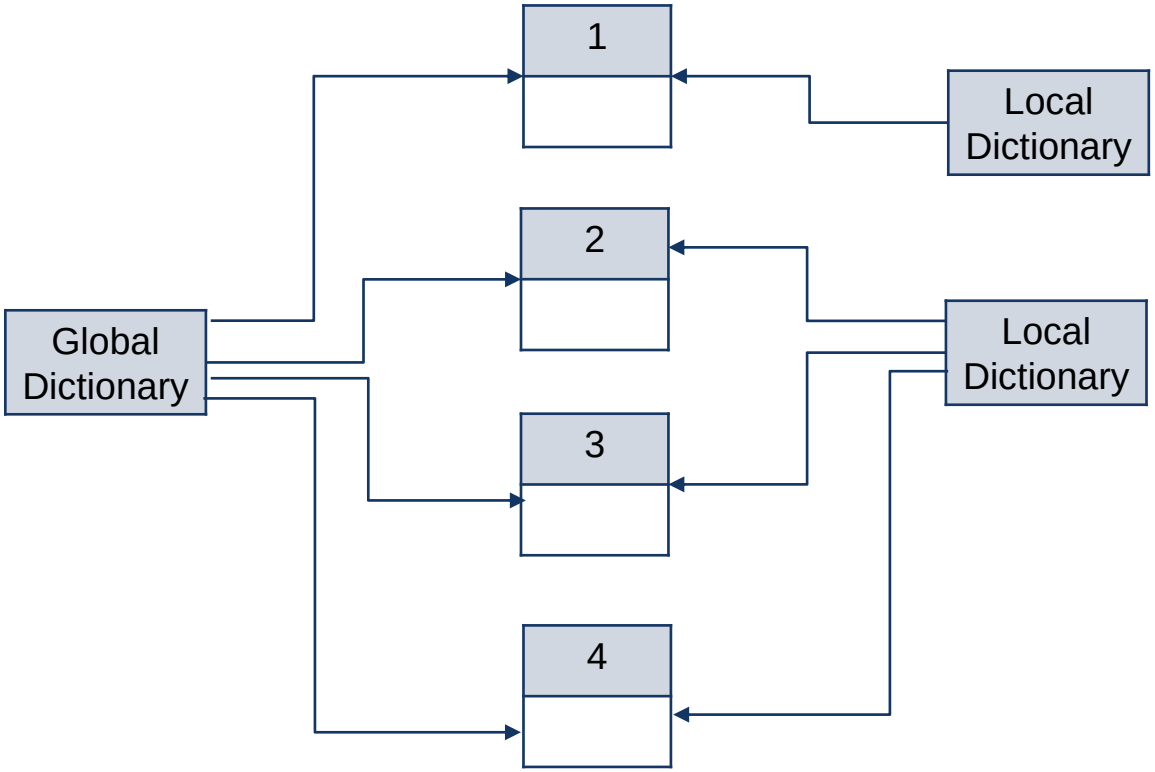
Compression Example:



For Columnstore

DICTIONARIES

Dictionaries types



Dictionaries

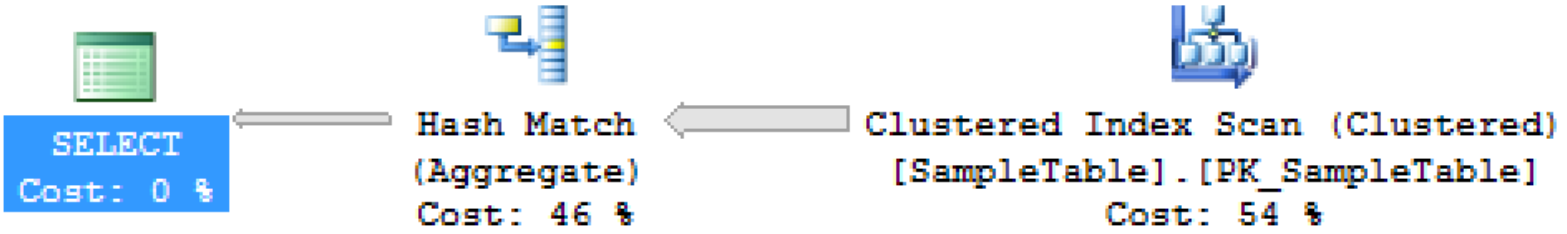
- *Global dictionaries*, contain entries for each and every of the existing segments of the same column storage
- *Local dictionaries*, contain entries for 1 or more segments of the same column storage
- Sizes varies from **56 bytes** (min) to 16 MB (max)
- There is a specialized view which provides information on the dictionaries, such as entries count, size, etc - sys.column_store_dictionaries
- Undocumented feature which potentially allow us to consult the content of the dictionaries (will see it later)
- No all columns will use dictionaries

MATERIALISATION

Let’s execute a query:

```
select name, count(*)
    from dbo.SampleTable
    group by name
    order by count(*) desc;
```

Execution Plan:



Materialisation Process

Name
Andre
Miguel
Sofia
Joana
Andre
Sofia
Vitor
Paulo
Joana
Miguel
Paulo
Paulo

Name	Value
Andre	1
Joana	2
Miguel	3
Paulo	4
Sofia	5
Vitor	6

Compressed
1
3
5
2
1
5
6
4
2
3
4
4

Select, Count, Group, Sort

Compressed
1
3
5
2
1
5
6
4
2
3
4
4

Item	Count
1	2
3	2
5	2
2	2
6	1
4	3

Item	Count
4	3
3	2
5	2
2	2
1	2
6	1

Name	Count
Andre	3
Joana	2
Miguel	2
Paulo	2
Sofia	2
Vitor	1

Meta-information

- sys.column_store_dictionaries – SQL Server 2012
- sys.column_store_segments – SQL Server 2012
- sys.column_store_row_groups – SQL Server 2014

Never try this on anything else besides your own test PC

DBCC CSINDEX

DBCC CSINDEX

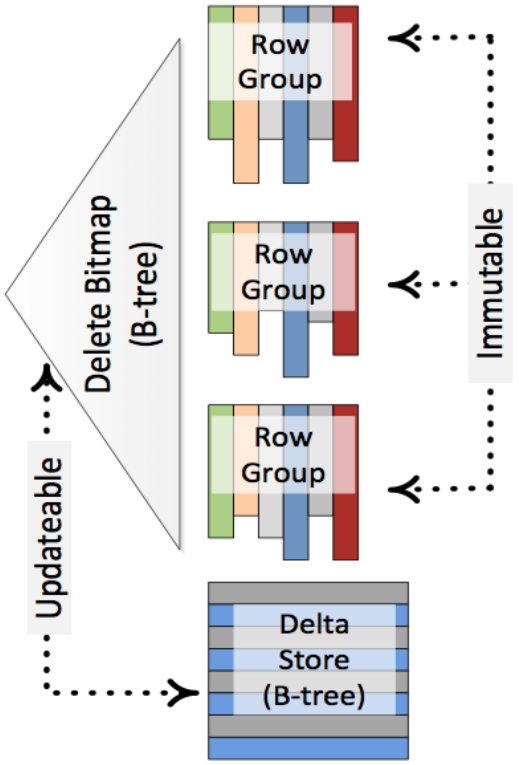
DBCC CSIndex (
{'dbname' | **dbid**},
rowsetid, --HoBT or PartitionID
columnid, -- Column_id from sys.column_store_segments
rowgroupid, -- segment_id from sys.column_store_segments
object_type, -- 1 (Segment), 2 (Dictionary),
print_option -- [0 or 1 or 2]
[, **start**]
[, **end**]
)

Clustered Columnstore Indexes

LOCKING

Columnstore elements:

- Row
- Column
- Row Group
- Segment
- Delta-Store
- Deleted Bitmap



But lock is placed

on Row Group/Delta-Store level

Columnstore

BULK LOAD

BULK Load

- A process completely apart
- 102.400 is a magic number which gives you a Segment instead of a Delta-Store
- For data load, if you order your loaded data into chunks of 1.045.678 rows for loading – your Columnstore will be almost perfect ☺

Columnstore Indexes

MEMORY MANAGEMENT

Memory Management

- Columnstore Indexes consume A LOT of memory
- Columnstore Object Pool – new special pool in SQL 2012+
- New Memory Brocker which divides memory between Row Store & Column Store

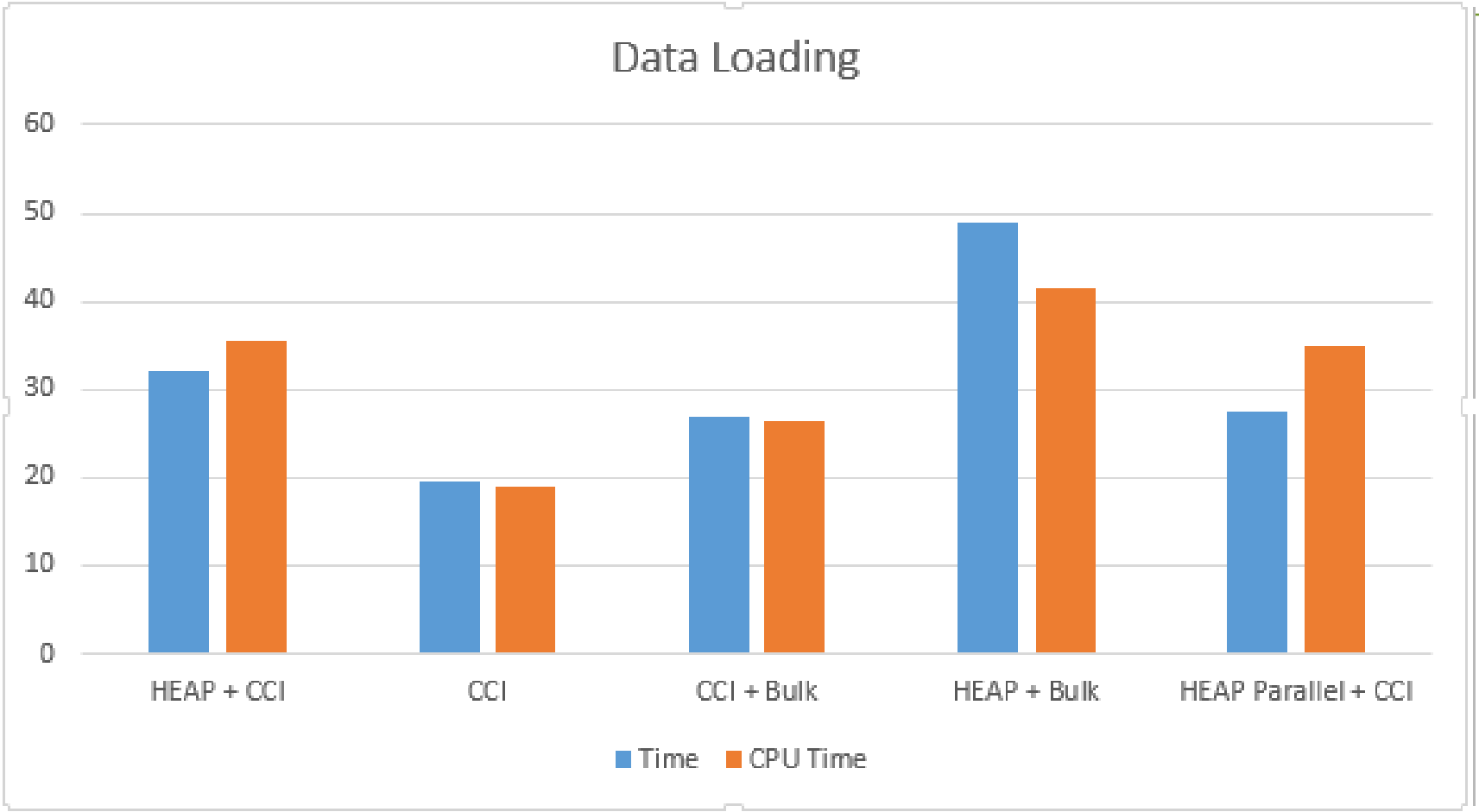
Memory Management

- Memory Grant for Index Creation in MB = (4.2 * Cols_Count + 68) * DOP + String_Cols * 34 **(2012 Formula)**
- When not enough memory granted, you might need to change Resource Governor limits for the respective group (here setting max percent grant to 50%):

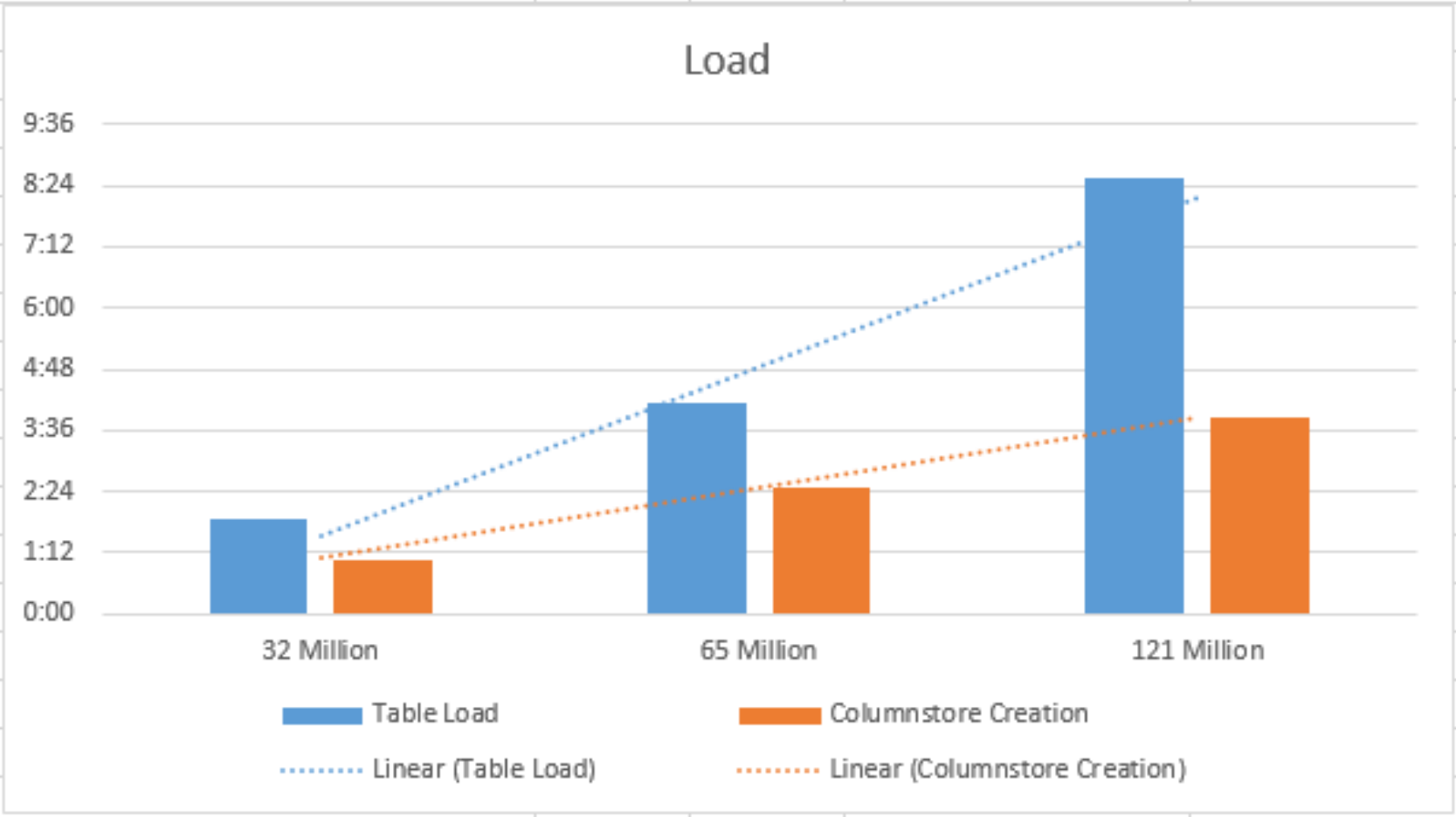
```
ALTER WORKLOAD GROUP [DEFAULT] WITH
(REQUEST_MAX_MEMORY_GRANT_PERCENT=50);
GO
ALTER RESOURCE GOVERNOR
            RECONFIGURE
GO
```

- Memory Management is automatic, so when you have not enough memory – then the DOP will be lowered automatically until 2, so the memory consumption will be lowered.

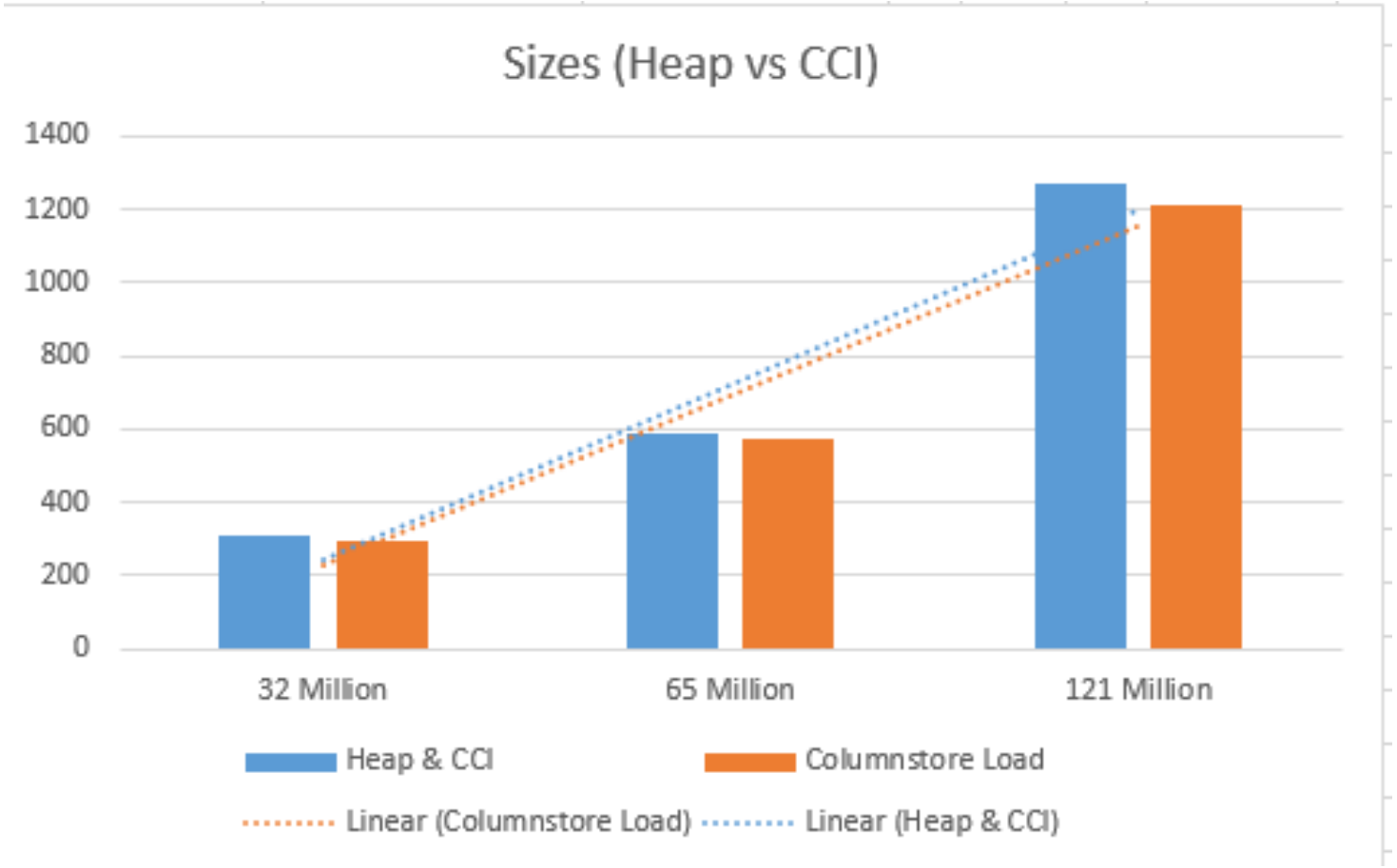
Data Loading



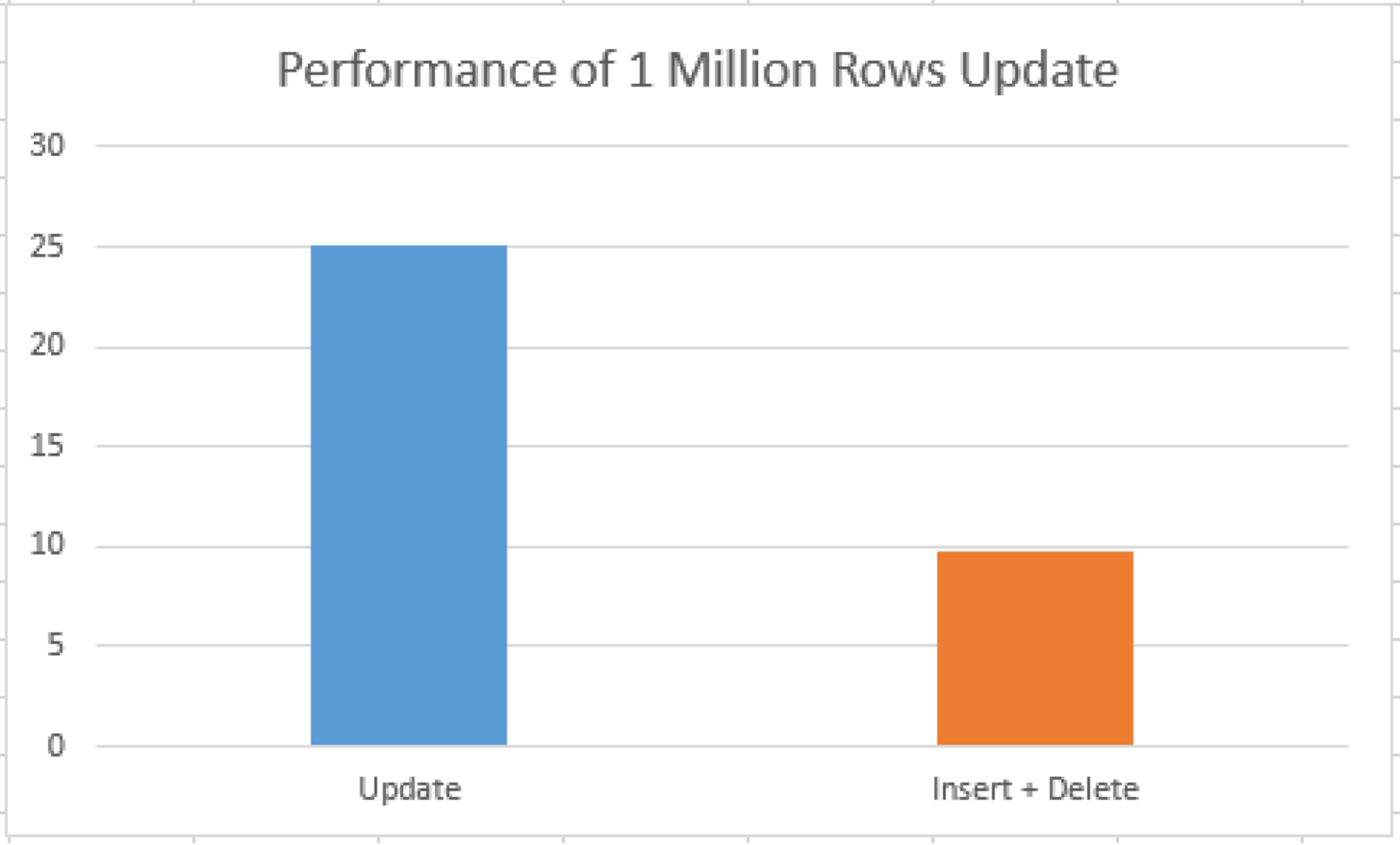
Data Loading



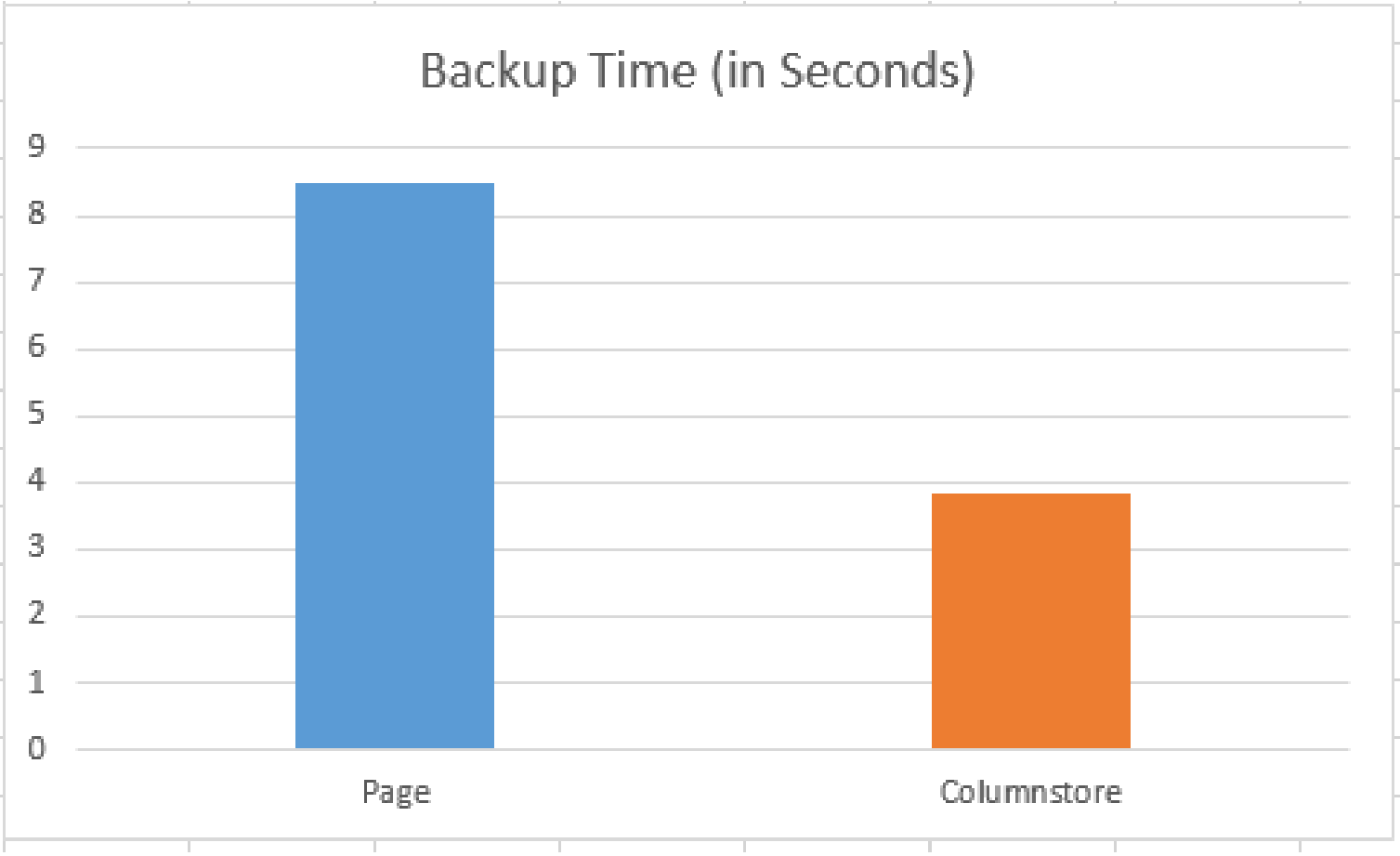
Data Loading



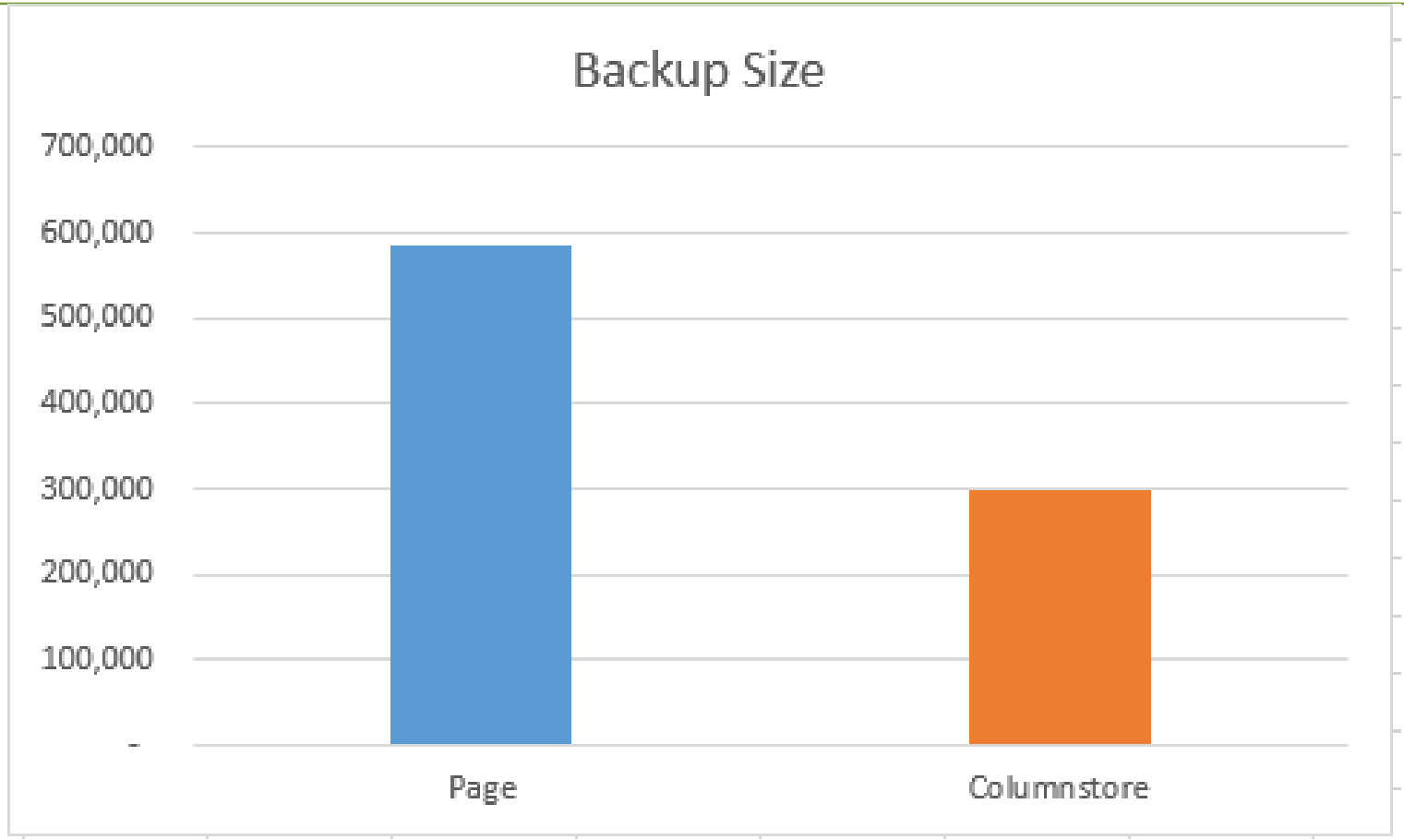
Update vs Delete + Insert



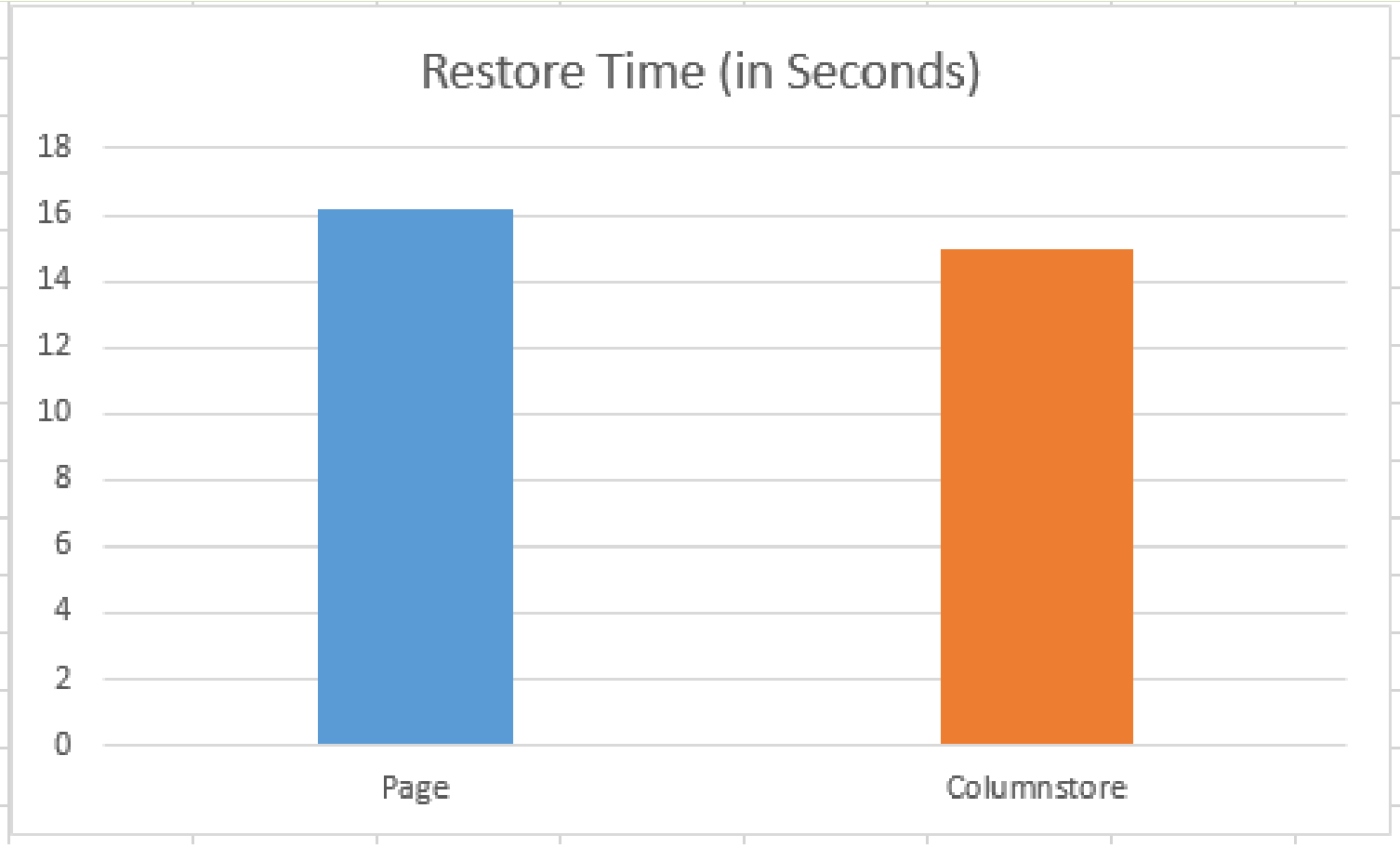
Backup



Backup size


Backup Size
700,000
600,000
500,000
400,000
300,000
200,000
100,000
-
Page
Columnstore

Restore


Restore Time (in Seconds)
18
16
14
12
10
8
6
4
2
0
Page
Columnstore

Muchas Gracias

Our Main Sponsors:

PERSONA CIÈNCIA EMPRESA

Universitat Ramon Llull

Microsoft

ApexSQL

SolidQ

ATTUNITY

Links:

My blog series on Columnstore Indexes (39+ Blogposts):

- http://www.nikoport.com/columnstore/

Remus Rusanu Introduction for Clustered Columnstore:

- http://rusanu.com/2013/06/11/sql-server-clustered-columnstore-indexes-at-teched-2013/

White Paper on the Clustered Columnstore:

- http://research.microsoft.com/pubs/193599/Apollo3%20-%20Sigmod%202013%20-%20final.pdf