SQL Server 2014 In-Memory Tables (Extreme Transaction Processing)

Advanced

Tony Rogerson, SQL Server MVP
@tonyrogerson
tonyrogerson@torver.net
http://www.sql-server.co.uk

Who am I?

- Freelance SQL Server professional and Data Specialist
- Fellow BCS, MSc in BI, PGCert in Data Science
- Started out in 1986 – VSAM, System W, Application System, DB2, Oracle, SQL Server since 4.21a
- Awarded SQL Server MVP yearly since 97
- Founded UK SQL Server User Group back in ’99, founder member of DDD, SQL Bits, SQL Relay, SQL Santa and Project Mildred
- Interested in commodity based distributed processing of Data.
- I have an allotment where I grow vegetables - I do not do any analysis on how they grow, where they grow etc. My allotment is where I escape technology.

Agenda

- Define concept “In-Memory”
- Test Harness
- Storage
- Row Chains
- Indexing
  - Hash
  - B^w Tree and B^+ Tree
  - Multi-Version Concurrency Control (MVCC)
- Monitoring

Define

“In-Memory”

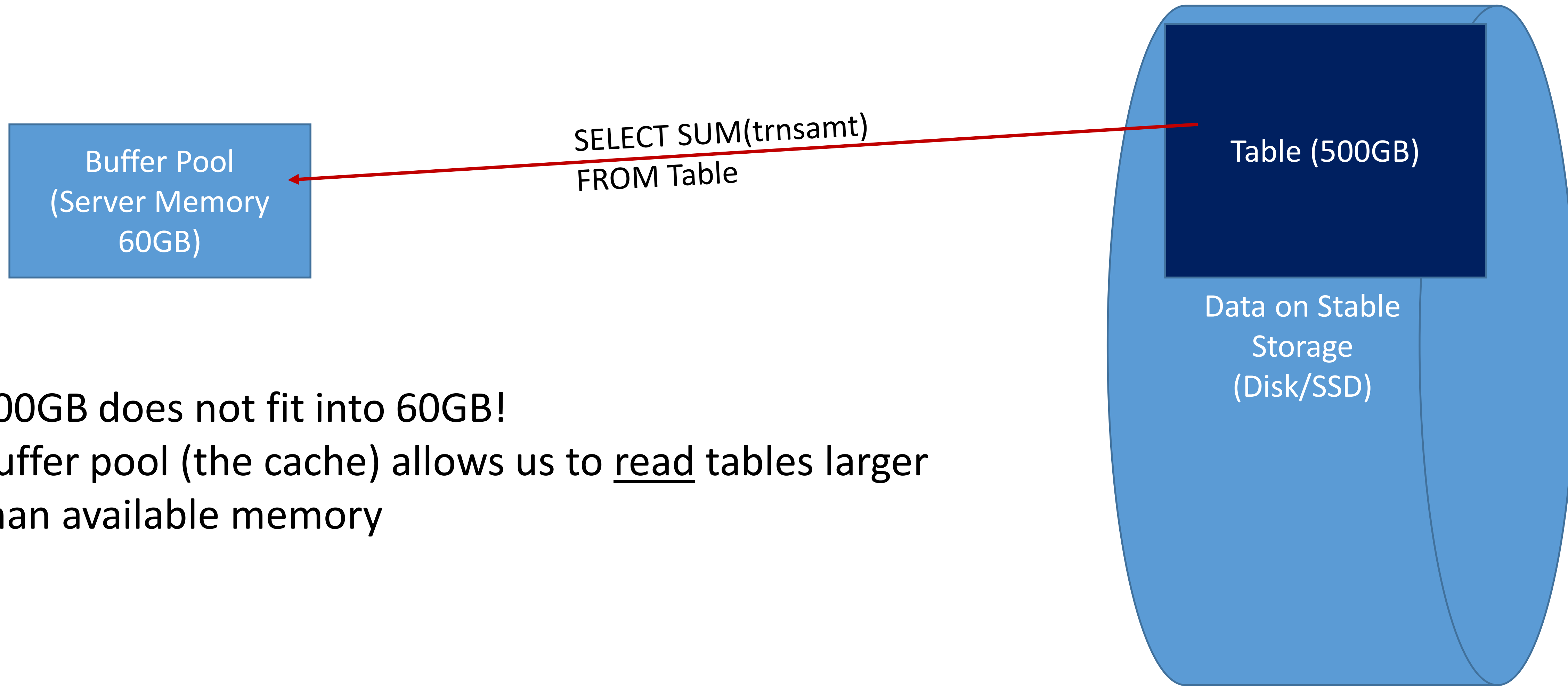
What is IMDB / DBIM?

- Entire database resides in Main memory
- Hybrid - selective tables reside entirely in memory
- Row / Column store
  - Oracle keeps two copies of the data – in row AND column store – optimiser chooses
  - SQL Server 2014 you choose –
    - Columnstore (Column Store Indexing) – traditionally for OLAP
    - Rowstore (memory optimised table with hash or range index) – traditionally for OLTP
- Non-Volatile Memory changes everything – it’s here now!
  - SQL Server 2014 Buffer Pool Extensions

SQL Server “in-memory”

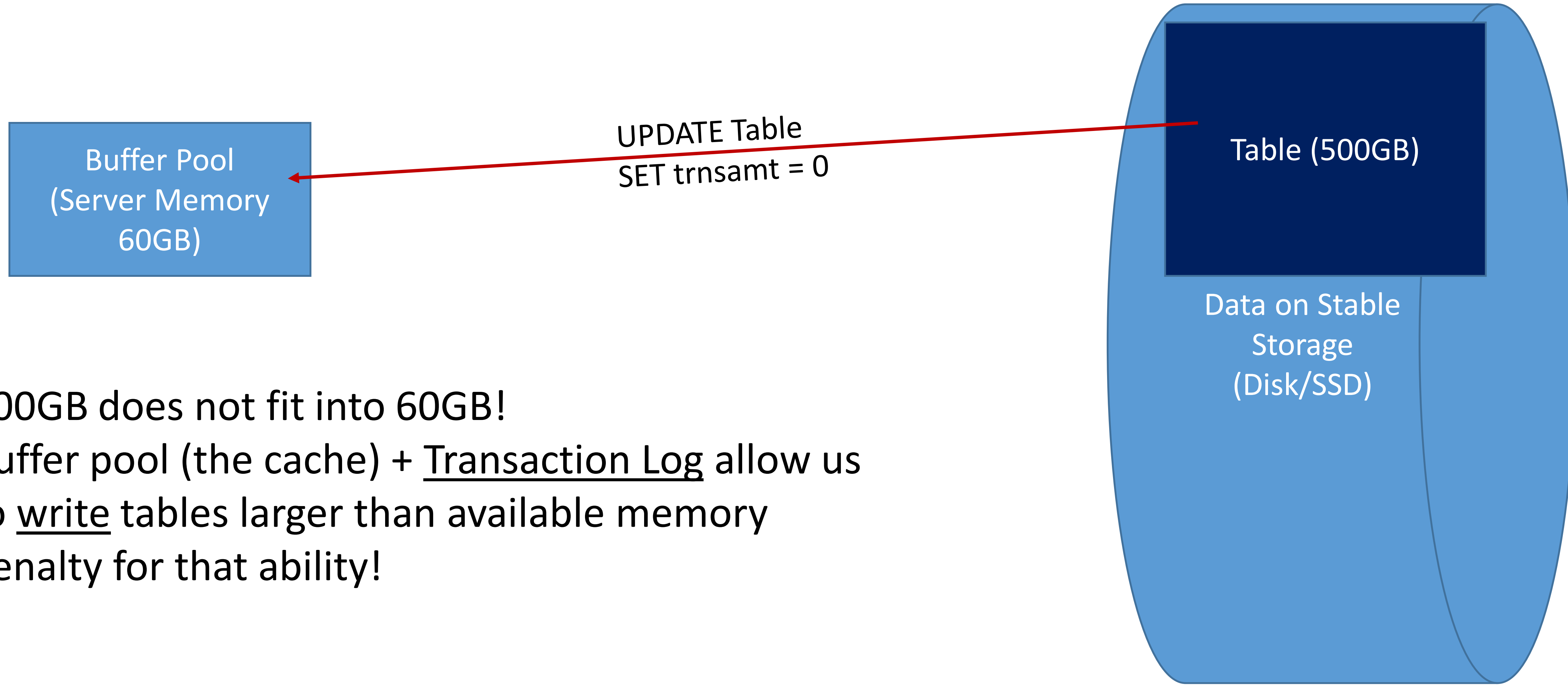
- Column Store (designed for OLAP)
  - Column compression to reduce memory required
  - SQL 2012:
    - One index per table, nonclustered only, table read-only
  - SQL 2014:
    - Only index on table – clustered, table is now updateable
- Memory Optimised Tables (designed for OLTP)
  - Standard CREATE TABLE with restrictions on FK’s and other constraints

Traditional Tables: Buffer Pool usage (Read)



- 500GB does not fit into 60GB!
- Buffer pool (the cache) allows us to read tables larger than available memory

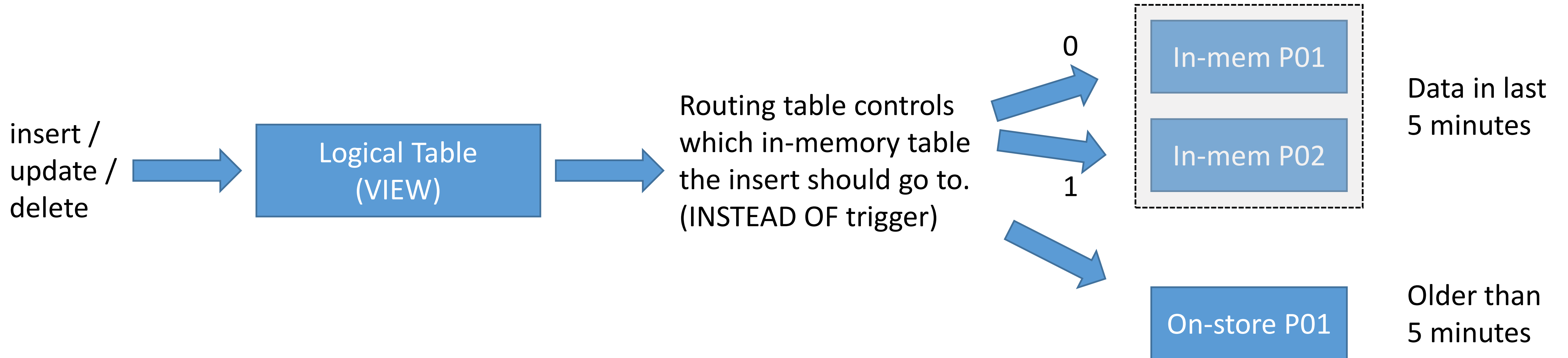
Traditional Tables: Buffer Pool usage (Write)



- 500GB does not fit into 60GB!
- Buffer pool (the cache) + Transaction Log allow us to write tables larger than available memory
- Penalty for that ability!

Test Harness

Test Harness



Test harness operations

- 10 connections each inserting 500 rows in a single transaction, running every 15 seconds with a random delay of 0 – 10 seconds.
- Manual Runs:
  - Move rows older than 5 minutes from in-memory into on-storage tables
  - Update rows

DEMO

- CREATEDB.sql
- PARTITIONING.sql
- EXEC.sql
- AGENT JOBS.sql (enable the jobs)

Storage

DEMO

- TRANSACTION LOG.sql

Transaction Logging

- All data is “in-memory” – WAL not required
- On COMMIT TRAN
  - Checks made depending on Isolation
  - Data made Durable (to Checkpoint File Pair (CFP))
- Versions of rows kept in memory (see MVCC)
- COMMIT is dramatically quicker because
  - Less data logged
    - no UNDO
    - no INDEX records
    - multiple log records combined grouped together to reduce log header overhead
  - Bulked IO

Storage

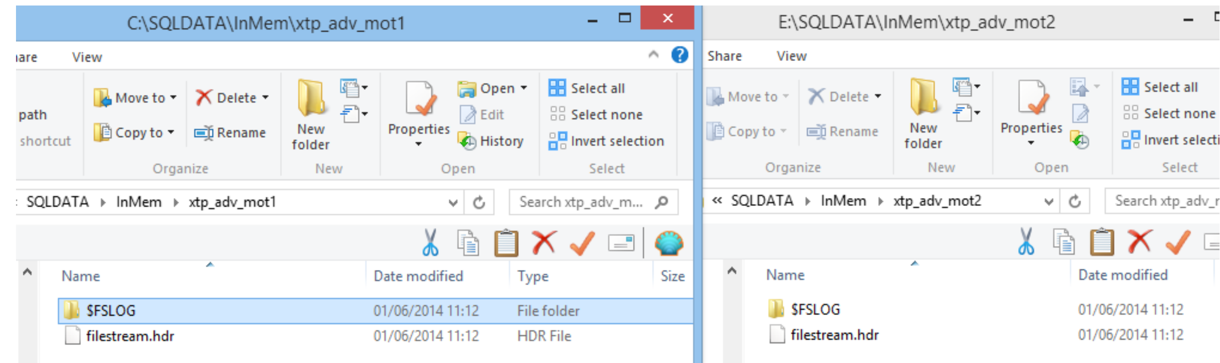
- System tables used for “some” MOT meta-data
- MOT have their own set of DMV’s

Traditional “on-storage” tables

xtp_adv_Data.mdf

xtp_adv_log.ldf

Memory Optimised tables
(CFP – Checkpoint File Pairs)



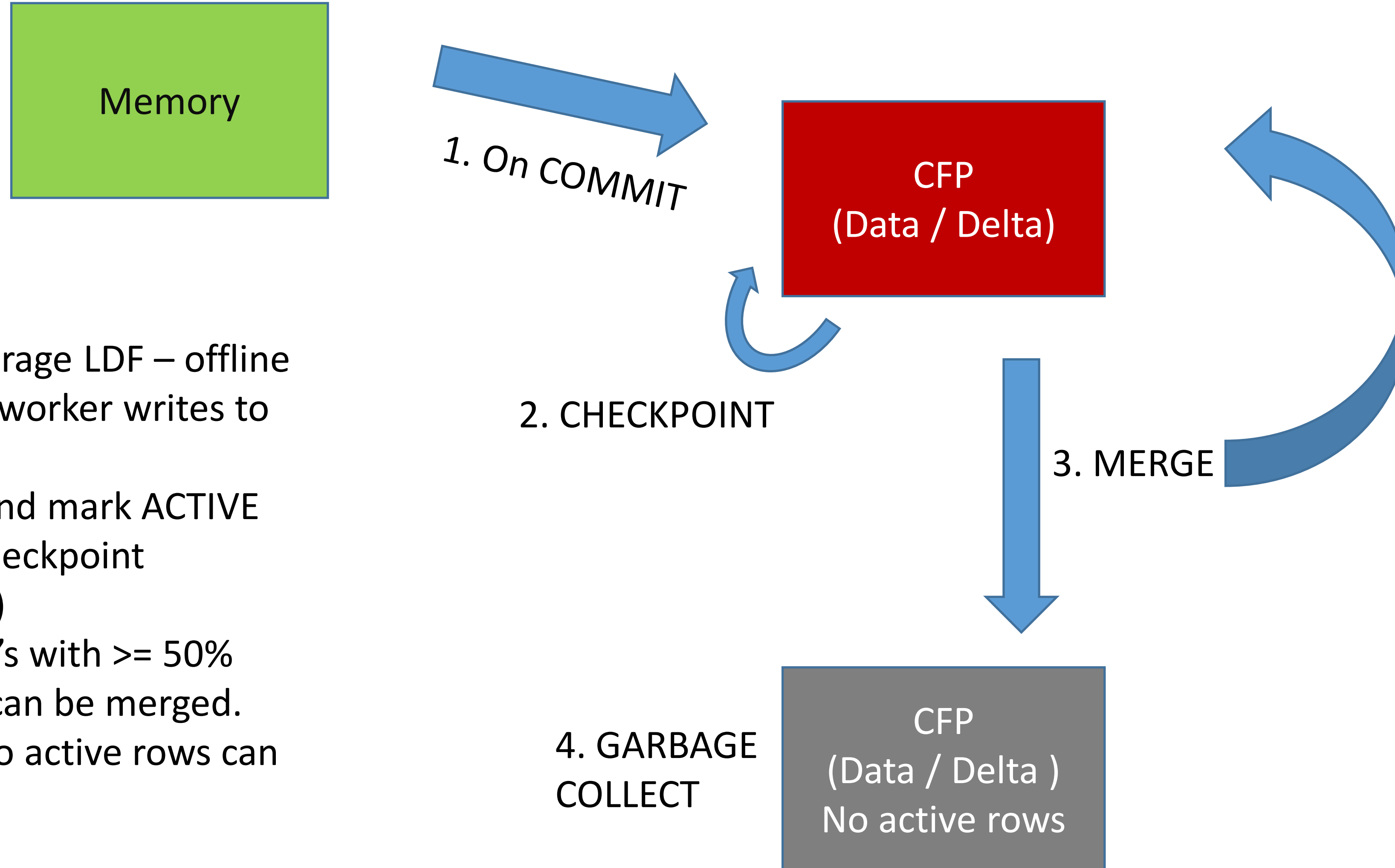
Storage of meta-data and compiled version of Native Compiled Procs and Tables.

Local Disk (C:) › Program Files › Microsoft SQL Server › MSSQL12.SQL2014 › MSSQL › DATA › xtp

How MOT use FILESTREAM

- Multiple Files
  - Data >= 16MiB or 128MiB files, stores row data
  - Delta >= 1MiB or 8MiB stores tombstone information
  - *Data/Delta may be larger if long running transactions*
- All reads and writes are <u>sequential</u>
- Offline Checkpoint Worker
  - Reads log
  - Appends to Data / Delta files
  - Random IO is not required!

Life of a Row Version



1. Write to storage LDF – offline checkpoint worker writes to CFP
2. Close CFP and mark ACTIVE (Durable Checkpoint established)
3. ACTIVE CFP's with >= 50% free space can be merged.
4. Files with no active rows can be deleted

CHECKPOINT

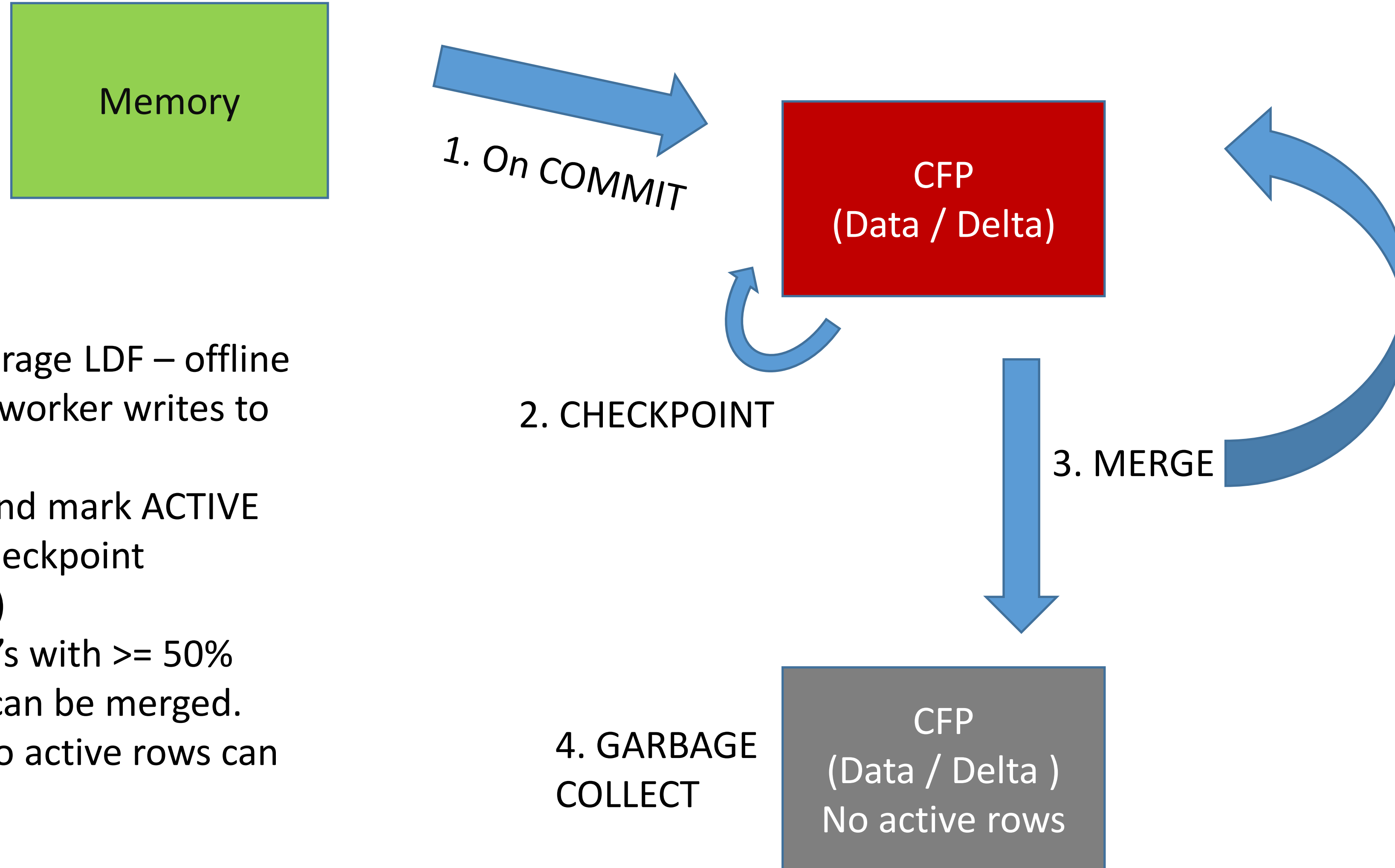
- After 512MiB data written to log or Manual CHECKPOINT
- CFP state: UNDER CONSTRUCTION (on recovery, data taken from transaction log)
- On CHECKPOINT
  - UNDER CONSTRUCTION CTP closed, becomes ACTIVE
  - Now have a durable checkpoint

DEMO

- CFP.sql

```
sys.dm_db_xtp_checkpoint_files
```

Life of a Row Version



1. Write to storage LDF – offline checkpoint worker writes to CFP
2. Close CFP and mark ACTIVE (Durable Checkpoint established)
3. ACTIVE CFP’s with >= 50% free space can be merged.
4. Files with no active rows can be deleted

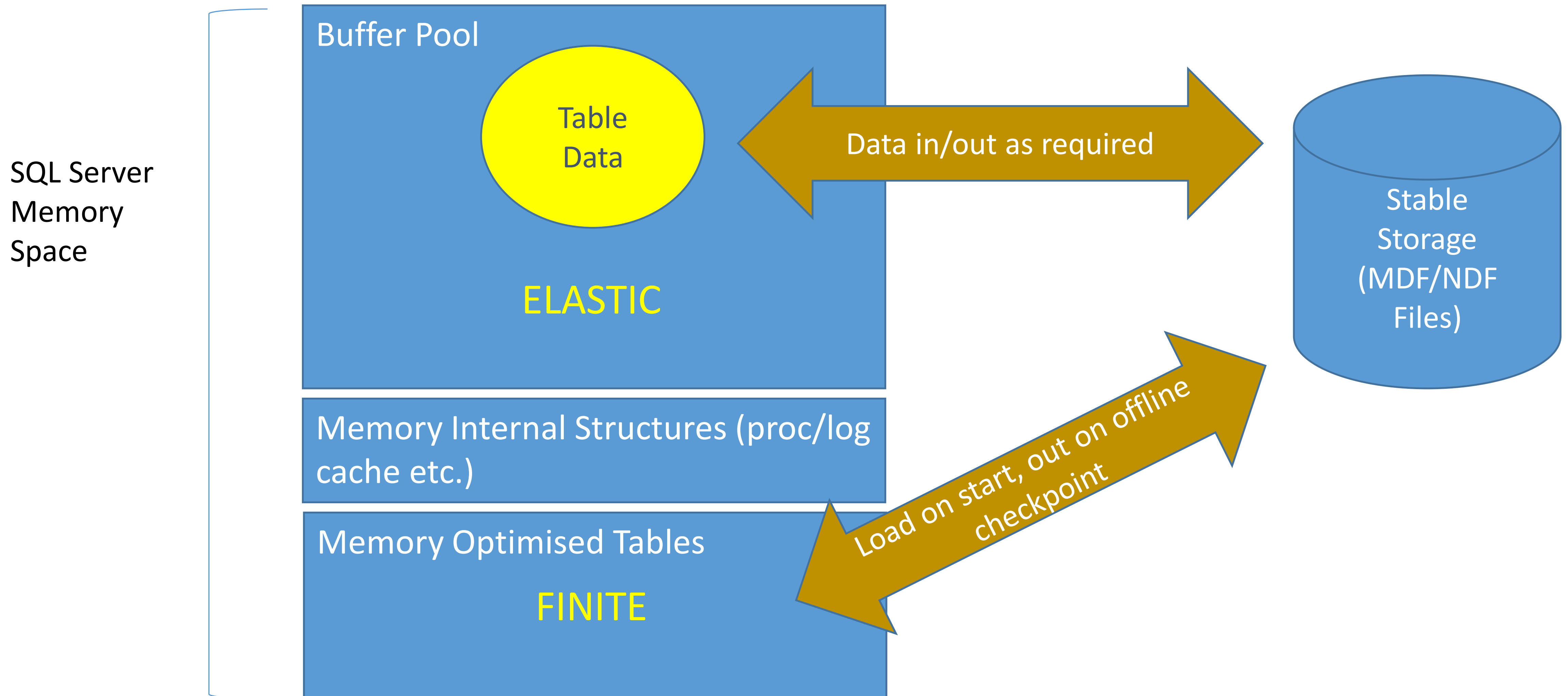
Row Updates

- Actually DELETE and INSERT
- Delete marker (not the row) is stored in DELTA file
- INSERT is stored in DATA file
- Versions hang around until no other sessions require them

Garbage Collection

- Before GC can be performed:
  - Rows get Durabilised ☺ to storage on COMMIT
  - Checkpoint File Pairs grow, new ones created
  - Various CFP will contain the “current” rows we want in case of a failure [remember – log gets backed up and truncated]
  - Merge CFP into a new one – pull out current rows into a new CFP
- CFP end up containing just old row versions no longer required
- GC removes old CFP’s (FILESTREAM GC)

SQL Server Memory Pools



Memory Optimised Tables

Indexing and how it works

Hash Indexing

Col1	Col2	Col3	Col4	Col5	Col6
2343	23	76	3	1	2
8812	12	24	1	4	3
3321	55	54	9	3	1
4123	23	99	0	5	-6
5123	23	99	0	2	3

1

Hash Index Columns

Col1	Col2	Col3
4123	23	99

2

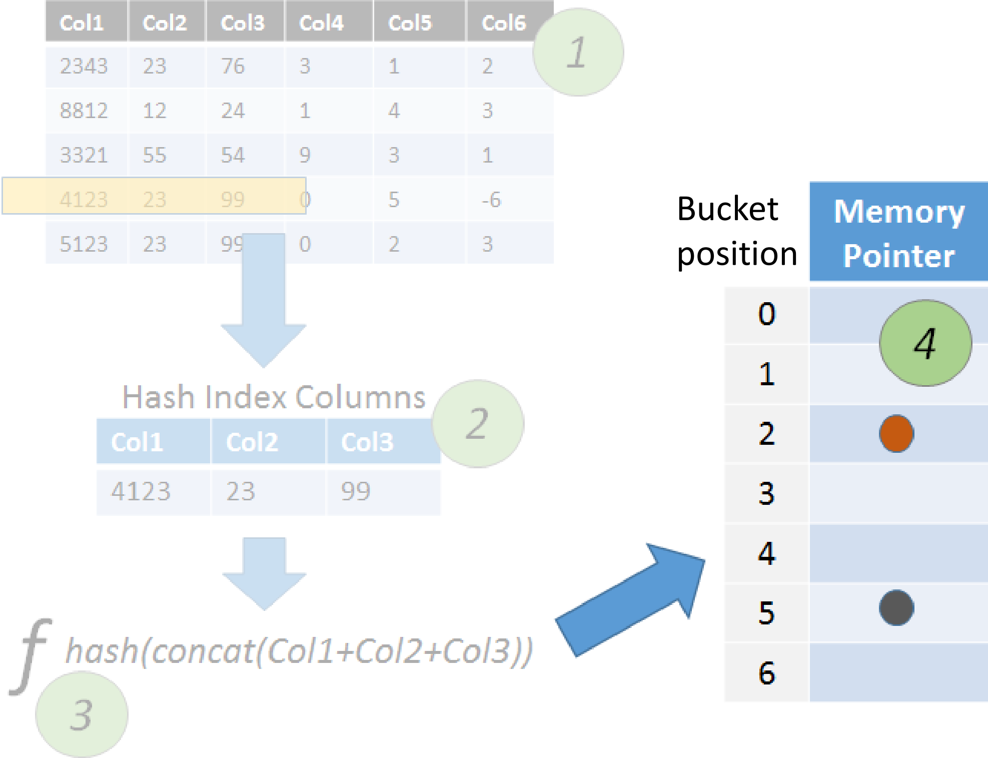
f *hash(concat(Col1+Col2+Col3))*

3

bucket_position = *f* hash(data)

Hash is deterministic (same value in gives same value out)

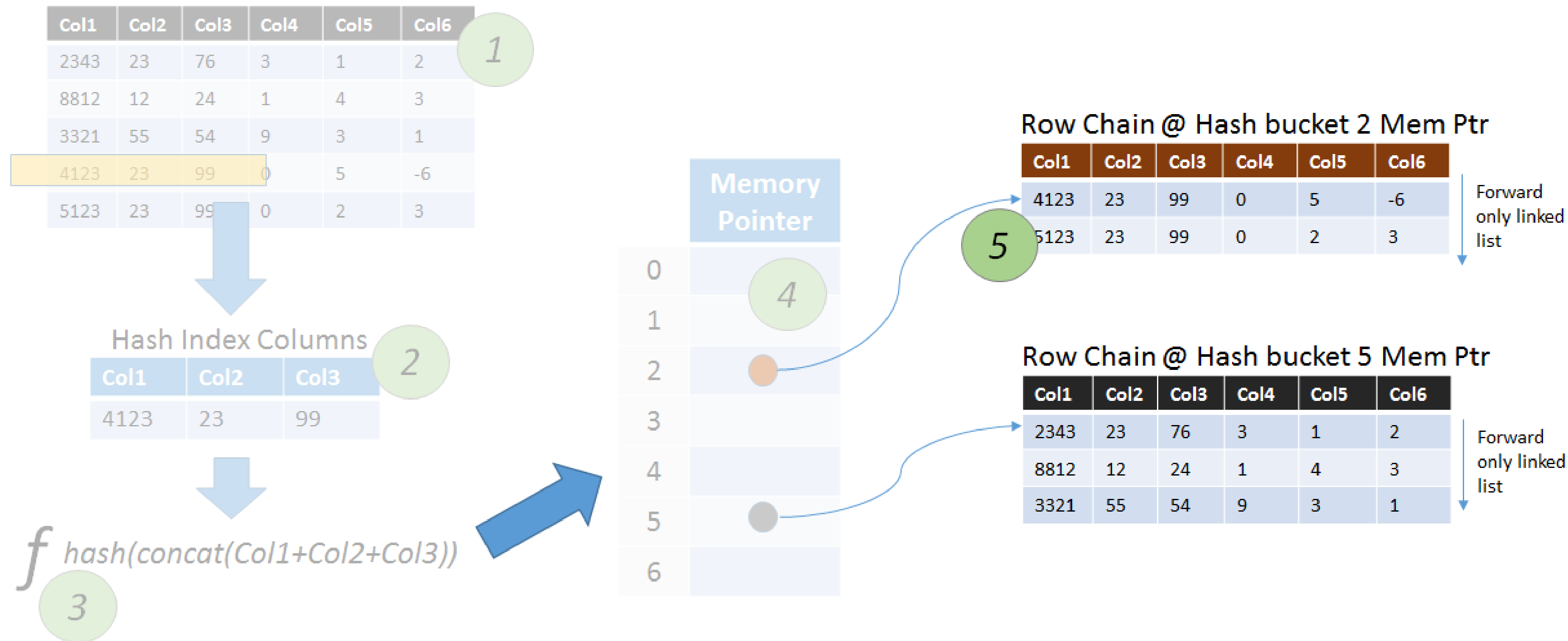
The hash is the position in the memory map table



Hash Collisions

- Example:1 million unique values hashed into 5000 hash values?
  - Multiple unique values mush hash to the same hash value
- Multiple key (real data) values hang off same hash bucket
- Termed: Row Chain
  - Chain of connected rows
- SQL Server 2014 Hash function is Balanced and follows a Poisson distribution – 1/3 empty, 1/3 at least one index key and 1/3 containing two index keys

Link from Memory Map Table to first row in Row Chain



Row Chain @ Hash bucket 2 Mem Ptr

Col1	Col2	Col3	Col4	Col5	Col6
4123	23	99	0	5	-6
5123	23	99	0	2	3

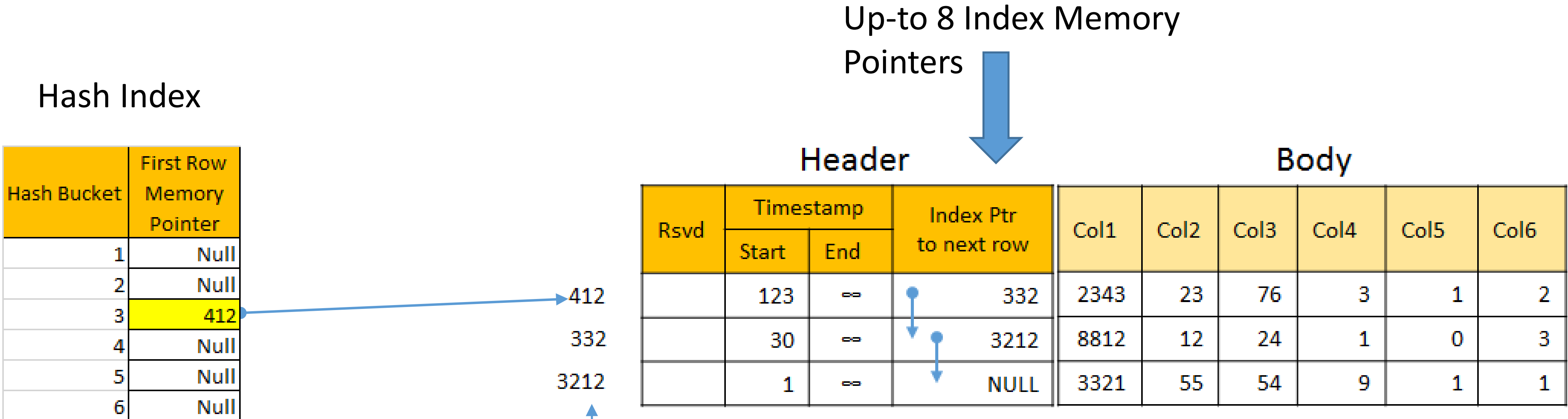
Forward only linked list

Row Chain @ Hash bucket 5 Mem Ptr

Col1	Col2	Col3	Col4	Col5	Col6
2343	23	76	3	1	2
8812	12	24	1	4	3
3321	55	54	9	3	1

Forward only linked list

Memory Optimised Table Row Header



- Above is a 3 row – row chain, all three rows hash to bucket #3
- Most recent row is first row in chain
- Rows in stable storage
  - accessed using files, pages and extents
- Rows in memory optimised
  - accessed using memory pointers

DEMO

- VISUALISE ROW CHAIN.sql

Row Header – Multiple Index Pointers

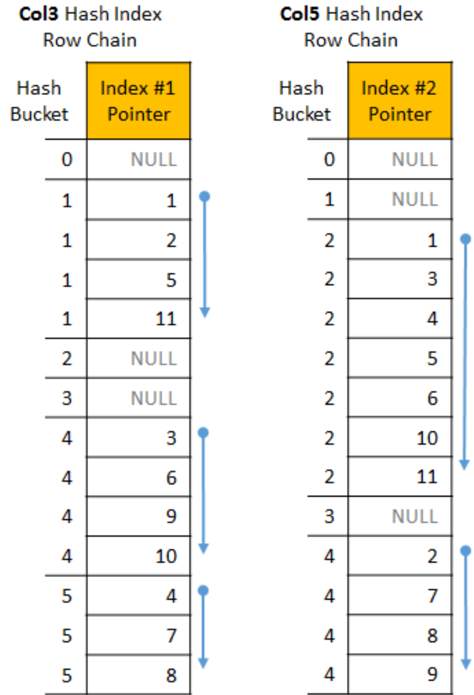
Data Rows (Index #‘*x*’ Pointer is <u>next</u> memory pointer in ‘row’ chain)

Memory Location	Col1	Col2	Col3	Col4	Col5	Col6	Index **Col3** Pointer	Index **Col5** Pointer
1	2343	23	76 **(1)**	3	1 **(2)**	2	2	3
2	8812	12	24 **(1)**	1	0 **(4)**	3	5	7
3	3321	55	54 **(4)**	9	1 **(2)**	1	6	4
4	3121	26	12 **(5)**	3	1 **(2)**	4	7	5
5	5545	25	33 **(1)**	3	1 **(2)**	2	11	6
6	2323	23	21 **(4)**	3	1 **(2)**	1	9	10
7	1123	21	45 **(5)**	3	0 **(4)**	4	8	8
8	1233	44	23 **(5)**	3	0 **(4)**	2	NULL	9
9	444	23	21 **(4)**	3	0 **(4)**	3	10	NULL
10	4434	11	65 **(4)**	3	1 **(2)**	2	NULL	11
11	7676	4	23 **(1)**	3	1 **(2)**	1	NULL	NULL

Value (*related hash_bucket*)

Row chain is defined through the Row Header

Memory Pointers per Hash Bucket



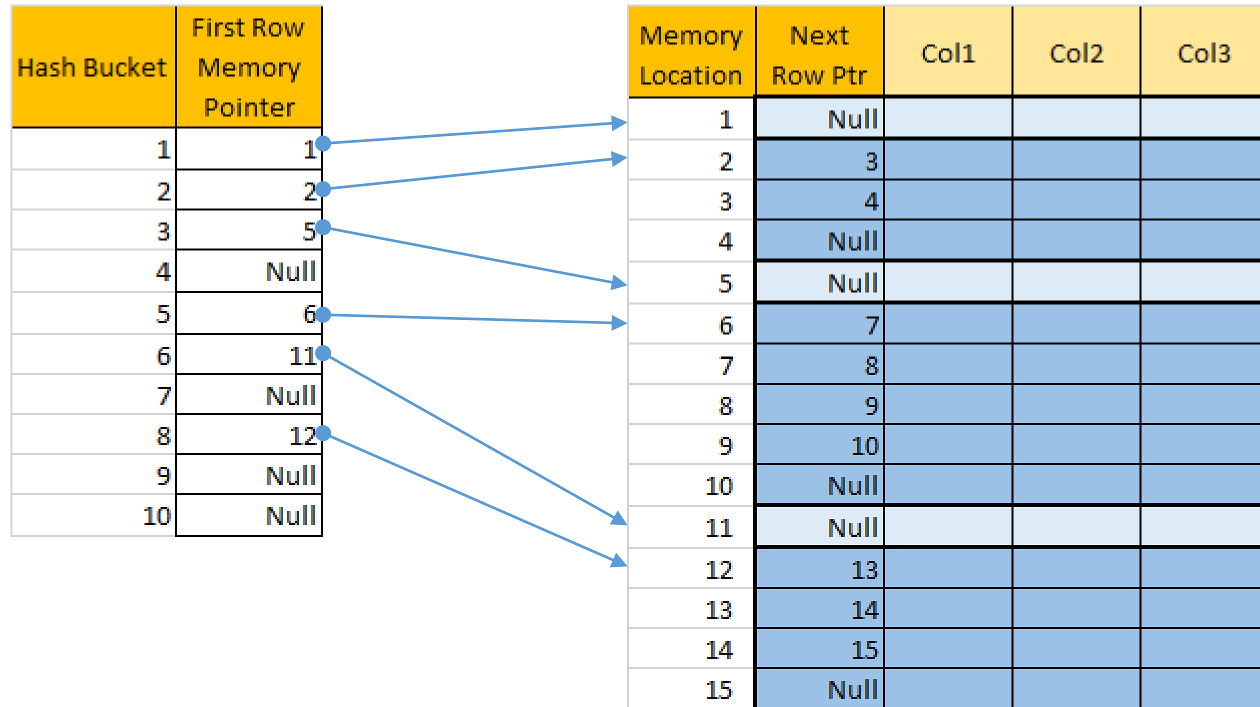
Col3 Hash Index Row Chain

Hash Bucket	Index #1 Pointer
0	NULL
1	1
1	2
1	5
1	11
2	NULL
3	NULL
4	3
4	6
4	9
4	10
5	4
5	7
5	8

Col5 Hash Index Row Chain

Hash Bucket	Index #2 Pointer
0	NULL
1	NULL
2	1
2	3
2	4
2	5
2	6
2	10
2	11
3	NULL
4	2
4	7
4	8
4	9

Hash Index Scan



Hash Bucket	First Row Memory Pointer
1	1
2	2
3	5
4	Null
5	6
6	11
7	Null
8	12
9	Null
10	Null

Memory Location	Next Row Ptr	Col1	Col2	Col3
1	Null			
2	3			
3	4			
4	Null			
5	Null			
6	7			
7	8			
8	9			
9	10			
10	Null			
11	Null			
12	13			
13	14			
14	15			
15	Null			

- Scan follows Hash Index (8 byte per bucket)
- Jump to each fist row in row chain
- Read row chain (lots of bytes per row – header + data)

BUCKET_COUNT options

1024 – 8 kilobytes
2048 – 16 kilobytes
4096 – 32 kilobytes
8192 – 64 kilobytes
16384 – 128 kilobytes
32768 – 256 kilobytes
65536 – 512 kilobytes

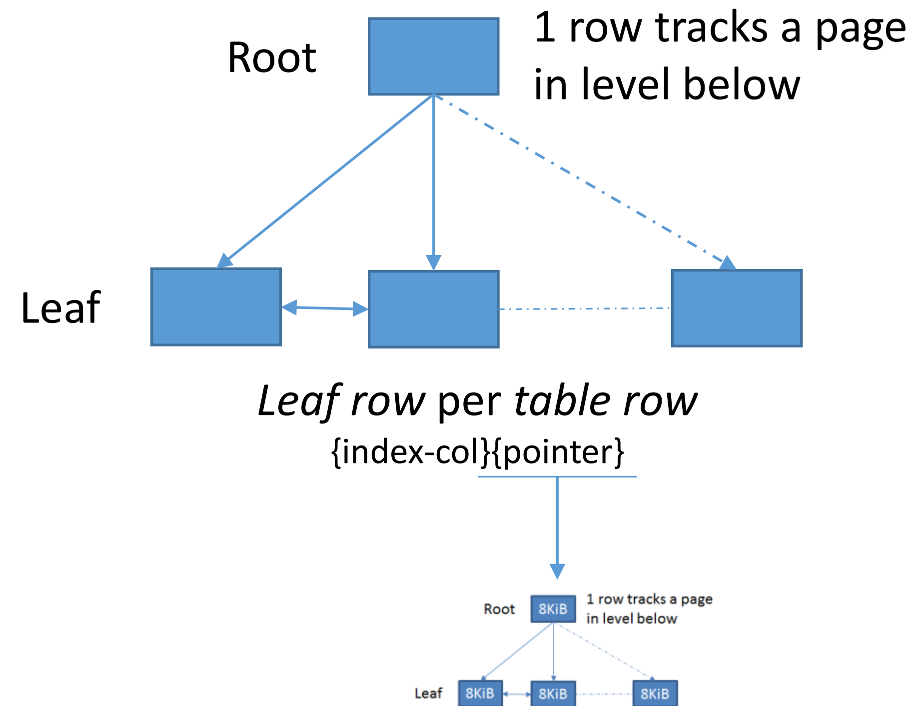
131072 – 1 megabytes
262144 – 2 megabytes
524288 – 4 megabytes
1048576 – 8 megabytes
2097152 – 16 megabytes
4194304 – 32 megabytes
8388608 – 64 megabytes

16777216 – 128 megabytes
33554432 – 256 megabytes
67108864 – 512 megabytes
134217728 – 1,024 megabytes
268435456 – 2,048 megabytes
536870912 – 4,096 megabytes
1073741824 – 8,192 megabytes

What makes a Row Chain grow?

- Hash collisions
- Row versions from updates
  - An update is a {delete, insert}
- Late garbage collection because of long running transactions
- Garbage collection can’t keep up because of box load and amount to do {active threads do garbage collection on completion of work}

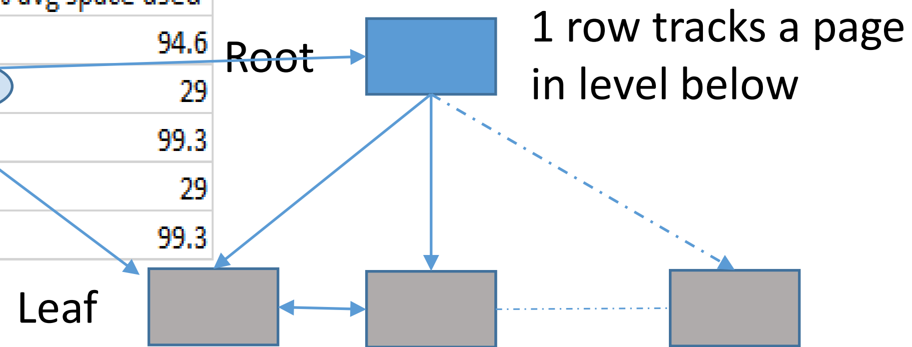
Range Index (B^W Tree ≈ B^+Tree)



Range Index (B^{W} Tree ≈ B^{+}Tree)

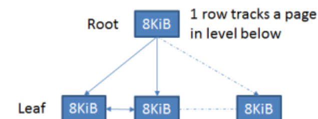
B^{+} Tree

index_id	name	index_level	page_count	record_count	% avg space used
0	NULL	0	111	50000	94.6
2	single_val	1	1	112	29
2	single_val	0	112	50000	99.3
3	unique_col	1	1	112	29
3	unique_col	0	112	50000	99.3



Leaf row per *table row*

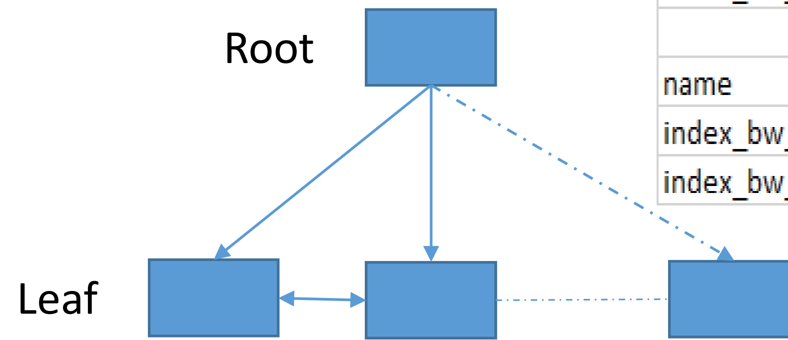
{index-col}{pointer}



Range Index (B^W Tree ≈ B^+Tree)

B^W Tree

name	table_used_memory_in_mb	table_unused_memory_in_mb
index_bw_tree_single	19.07	0.05
index_bw_tree_unique	19.07	0.05
name	index_used_memory_in_mb	index_unused_memory_in_mb
index_bw_tree_single	0	0.19
index_bw_tree_unique	6.12	5.88



Leaf row per *unique value*

{index-col}{memory-pointer}

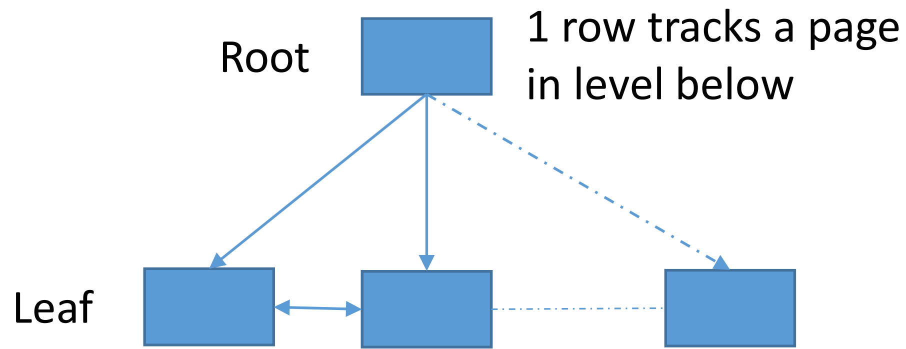
Row Chain

account_no	active
7	1
6	1
5	1
4	1
3	1
2	1
1	1

account_no	active
14	0
13	0
12	0
10	0

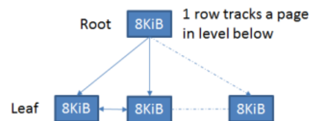
Range Index (B^W Tree ≈ B^+Tree)

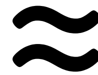
B^+Tree



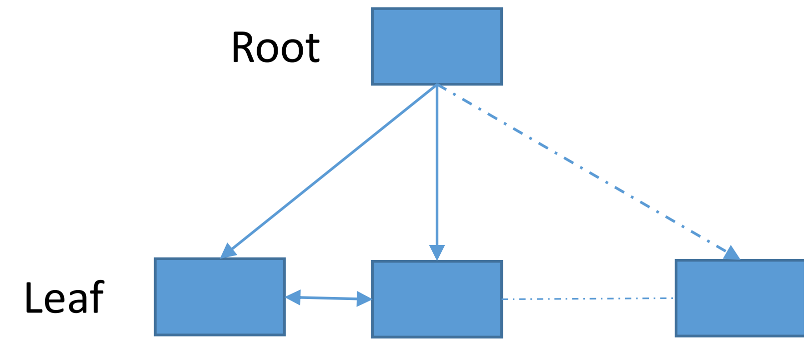
Leaf row per *table row*

{index-col}{pointer}





B^W Tree



Leaf row per *unique value*

{index-col}{memory-pointer}

Row Chain

account_no	active
7	1
6	1
5	1
4	1
3	1
2	1
1	1

account_no	active
14	0
13	0
12	0
10	0

Multi-Version Concurrency Control (MVCC)

- Compare And Swap replaces Latching
- Update is actually INSERT and Tombstone
- Row Versions [kept in memory]
- Garbage collection task cleans up versions no longer required (stale data rows)
- Version not required determined by active transactions and snapshot the connection has of the data
- Your in-memory table can double, triple, x 100 etc. in size depending on access patterns (data mods + query durations)

Versions in the Row Chain

	From Hash Bucket	TS_START	TS_END	FirstName	Surname	Pints Drank
Newest	23212	43	NULL	Mark	Whitehorn	7
	23212	26	42	Mark	Whitehorn	6
	23212	13	25	Mark	Whitehorn	5
	23212	9	12	Mark	Whitehorn	4
	23212	6	8	Mark	Whitehorn	3
	23212	3	5	Mark	Whitehorn	2
Oldest	23212	1	2	Mark	Whitehorn	1

TS_START	TS_END	Connection
2	NULL	55
7	NULL	60
50	NULL	80

Versions in the Row Chain

	From Hash Bucket	TS_START	TS_END	FirstName	Surname	Pints Drank
Newest	23212	43	NULL	Mark	Whitehorn	7
	23212	26	42	Mark	Whitehorn	6
	23212	13	25	Mark	Whitehorn	5
	23212	9	12	Mark	Whitehorn	4
Oldest	23212	6	8	Mark	Whitehorn	3
	23212	3	5	Mark	Whitehorn	2
	23212	1	2	Mark	Whitehorn	1

Tombstone

TS_START	TS_END	Connection
2	10	55
7	NULL	60
50	NULL	80

Versions in the Row Chain

	From Hash Bucket	TS_START	TS_END	FirstName	Surname	Pints Drank
Current	23212	43	NULL	Mark	Whitehorn	7
	23212	26	42	Mark	Whitehorn	6
	23212	13	25	Mark	Whitehorn	5
	23212	9	12	Mark	Whitehorn	4
	23212	6	8	Mark	Whitehorn	3
	23212	3	5	Mark	Whitehorn	2
	23212	1	2	Mark	Whitehorn	1

Tombstone

TS_START	TS_END	Connection
2	NULL	55
7	NULL	60
50	NULL	80

The problem with Versions

- Available memory is finite
- All data must fit in memory
- You may well run out of memory!!
- Removal of versions from memory is not immediate

Monitoring

- sys.hash_indexes
- sys.dm_db_xtp_hash_index_stats
- sys.dm_db_xtp_table_memory_stats
- sys.dm_db_xtp_index_stats
- sys.dm_db_xtp_object_stats

Summary

- Memory is finite
- Access patterns
  - Watch your long running transactions
  - COMMIT is now your friend (batch up)
  - Version creation is your enemy
- Consider Index strategy very careful and monitor