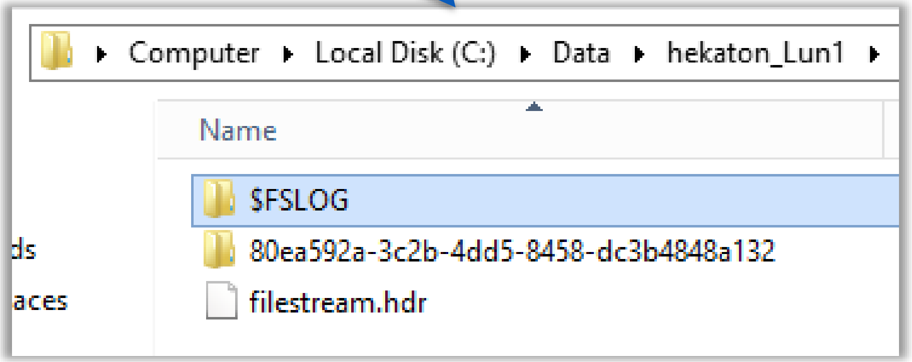
Create Filegroup

```
CREATE DATABASE [Hekaton]
ON PRIMARY
(NAME = N'Hekaton_data', FILENAME = N'C:\Data\Hekaton_data.mdf'),
FILEGROUP [Hekaton_InMemory] CONTAINS MEMORY_OPTIMIZED_DATA
(NAME = N'Hekaton_mem', FILENAME = N'C:\Data\Mem\Hekaton_Lun1')
LOG ON
(NAME = N'Hekaton_log', FILENAME = N'C:\Data\Log\Hekaton_log.ldf')
```

FileGroup Container

```
ALTER DATABASE [Hekaton]
ADD FILE (NAME = N'Hekaton_mem',
FILENAME = N'C:\Data\Mem\Hekaton_Lun2')
TO FILEGROUP [Hekaton_InMemory]
```


Computer ▸ Local Disk (C:) ▸ Data ▸ hekaton_Lun1 ▸
Name
$FSLOG
80ea592a-3c2b-4dd5-8458-dc3b4848a132
filestream.hdr
ds
aces

Create Memory Optimized Table

```
CREATE TABLE [Customer] (
 [CustomerID] INT NOT NULL
 PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT = 1000000),
 [AddressID] INT NOT NULL INDEX [IxName] HASH WITH (BUCKET_COUNT =
1000000),
 [LName] NVARCHAR(250) COLLATE Latin1_General_100_BIN2 NOT NULL
 INDEX [IXLName] NONCLUSTERED (LName)
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Hash Index

Collation BIN2

Range Index

This table is memory optimized

This table is durable

Memory Optimized Table Limitations

Optimized for high-throughput OLTP

- No DML triggers
- No XML and no CLR data types

Optimized for in-memory

- Rows are at most 8060 bytes – no off row data
- No Large Object (LOB) types like varchar(max)

Scoping limitations

- No FOREIGN KEY and no CHECK constraints
- No schema changes (ALTER TABLE) – need to drop/recreate table
- No add/remove index – need to drop/recreate table
- No Computed Columns
- No Cross-Database Queries

Data Access to Memory Optimized Tables

Interpreted T-SQL Access

- Access both memory and disk based tables
- Less performant
- Virtually full T-SQL surface
- **When to use**
  - Ad-hoc queries
  - Reporting-style queries
  - Speeding up app migration

Natively Compiled Procs

- Access only memory optimized tables
- Maximum performance
- Limited T-SQL surface area
- **When to use**
  - OLTP-style operations
  - Optimize performance critical business logic
  - More the logic embedded, better the performance improvement

Natively Compiled Procedures

```
CREATE PROCEDURE [dbo].[InsertOrder](@id INT, @date DATETIME)
WITH
NATIVE_COMPILATION,
SCHEMABINDING,
EXECUTE AS OWNER
AS
BEGIN ATOMIC
WITH
(TRANSACTION ISOLATION LEVEL = SNAPSHOT,
LANGUAGE = N'us_english')
      -- Insert T-SQL here…
END
```

This proc is natively compiled

Native procs must be schema-bound

Execution context is required

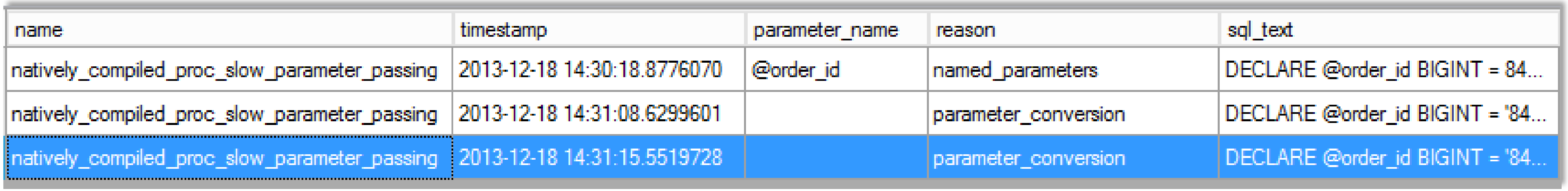
Atomic blocks
- Create a transaction if there is none
- Otherwise, create a savepoint

Session settings are fixed at create time

Natively Compiled Procedure Performance

To avoid the server having to map parameter names and convert types:

- Use ordinal (nameless) parameters, do not use named parameters
- Match parameter types passed.
- Use Extended Event ‘natively_compiled_proc_slow_parameter_passing’ to identify.



name	timestamp	parameter_name	reason	sql_text
natively_compiled_proc_slow_parameter_passing	2013-12-18 14:30:18.8776070	@order_id	named_parameters	DECLARE @order_id BIGINT = 84...
natively_compiled_proc_slow_parameter_passing	2013-12-18 14:31:08.6299601		parameter_conversion	DECLARE @order_id BIGINT = '84...
natively_compiled_proc_slow_parameter_passing	2013-12-18 14:31:15.5519728		parameter_conversion	DECLARE @order_id BIGINT = '84...

Memory Optimized TVP and Table Variables

Usage Scenarios

- Storing Intermediate Result Sets
- Replacement for table variables
- Passing TVP to Natively compiled procedures

Advantages

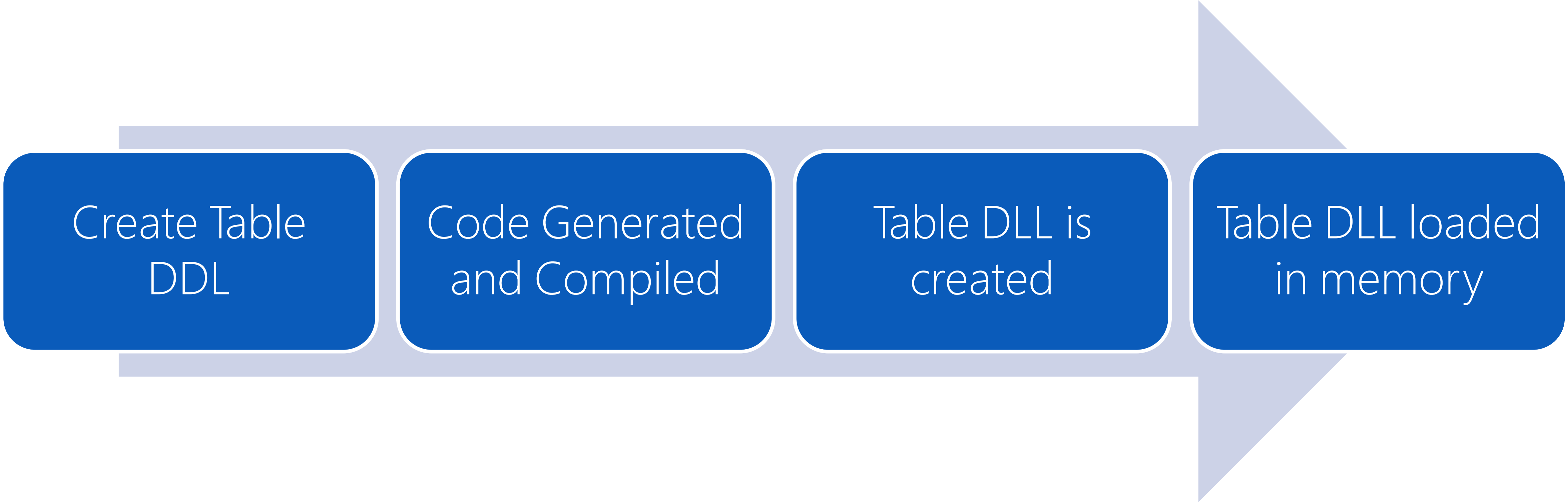
- Always in memory, never spill to disk
- No Tempdb utilization
- Data access is efficient

Memory Optimized TVP

```
CREATE TYPE InMemTVP AS TABLE
(ProductID INT NOT NULL PRIMARY KEY NONCLUSTERED HASH(ProductID)
WITH (BUCKET_COUNT = 100))
WITH (MEMORY_OPTIMIZED = ON)
GO

CREATE PROCEDURE [dbo].[GetProductDetails_Native](@tvp dbo.InMemTVP READONLY)
WITH
NATIVE_COMPILATION,
SCHEMABINDING,
EXECUTE AS SELF
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'English')
SELECT a.ProductID, Name, ListPrice FROM Production.Product_inmem a
INNER JOIN @tvp b ON a.ProductID = b.ProductID
END
GO
```

Memory Optimized Table Creation



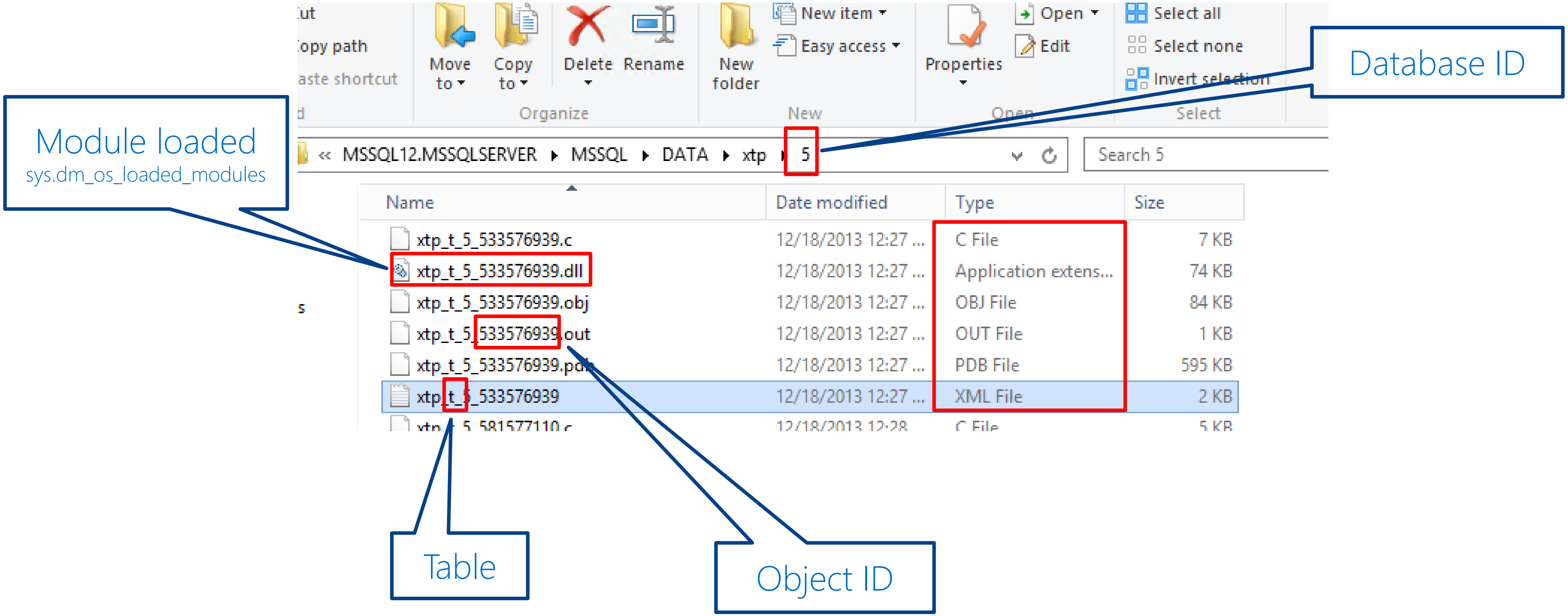
Create Table DDL

Code Generated and Compiled

Table DLL is created

Table DLL loaded in memory

Native Compilation Files



Name	Date modified	Type	Size
xtp_t_5_533576939.c	12/18/2013 12:27 ...	C File	7 KB
xtp_t_5_533576939.dll	12/18/2013 12:27 ...	Application extens...	74 KB
xtp_t_5_533576939.obj	12/18/2013 12:27 ...	OBJ File	84 KB
xtp_t_5_533576939.out	12/18/2013 12:27 ...	OUT File	1 KB
xtp_t_5_533576939.pdb	12/18/2013 12:27 ...	PDB File	595 KB
xtp_t_5_533576939	12/18/2013 12:27 ...	XML File	2 KB
xtp_t_5_581577110.c	12/18/2013 12:28	C File	5 KB

Database ID

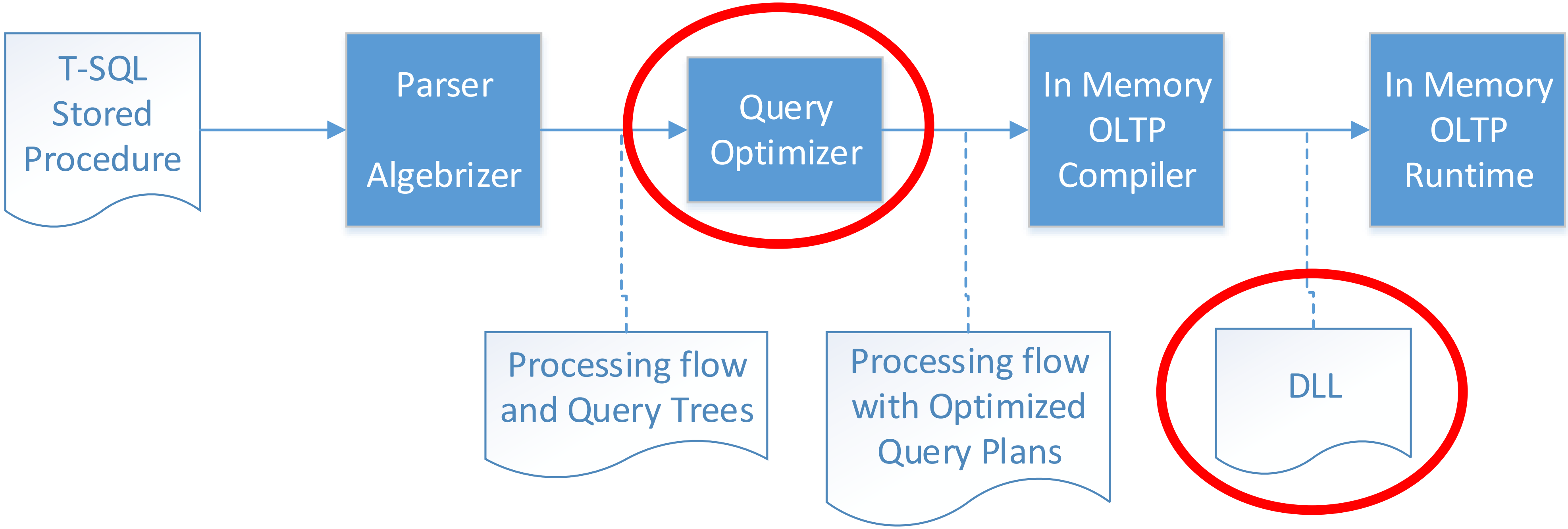
Module loaded
sys.dm_os_loaded_modules

Table

Object ID

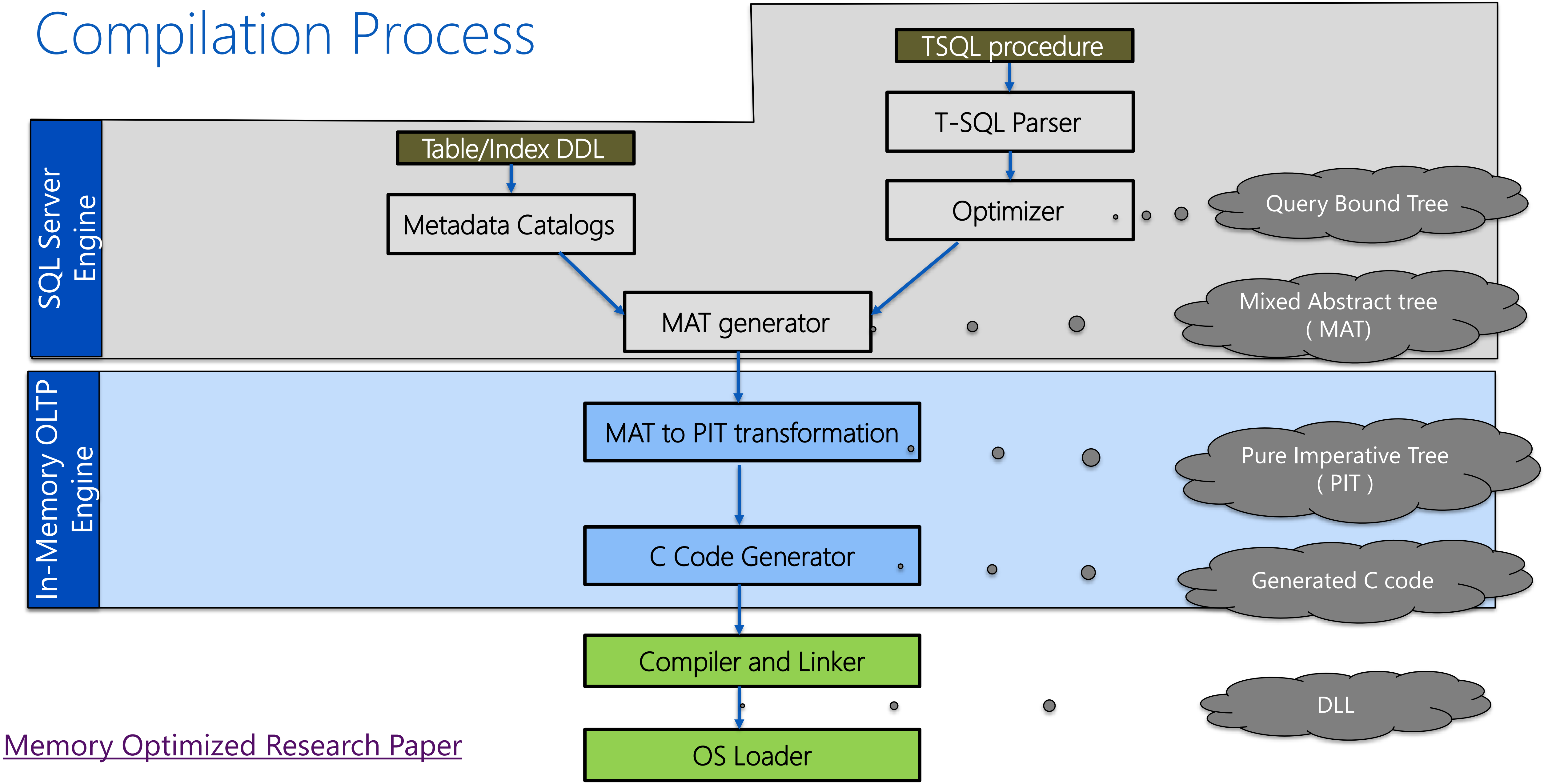
Natively Compiled Procedure

Queries are optimized at procedure create time



DLLs are **not** part of the plan cache

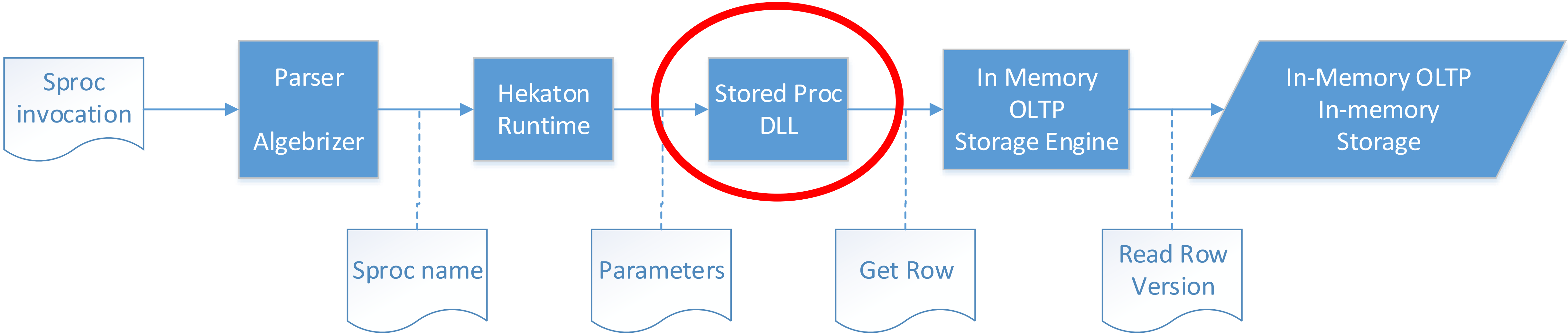
Compilation Process



Memory Optimized Research Paper

Natively Compiled Procedure Execution

Query operators and functions are baked into DLL



Other Considerations

Parameter sniffing

- For Natively compiled procedures, UNKNOWN parameter value assumed
- Still at play for Interpreted T-SQL

Recompilation for Natively compiled procedures

- No automatic recompilation, requires a drop and recreate
- Recompiled on first execution after restart of the Server or failover

In-Memory OLTP Structures

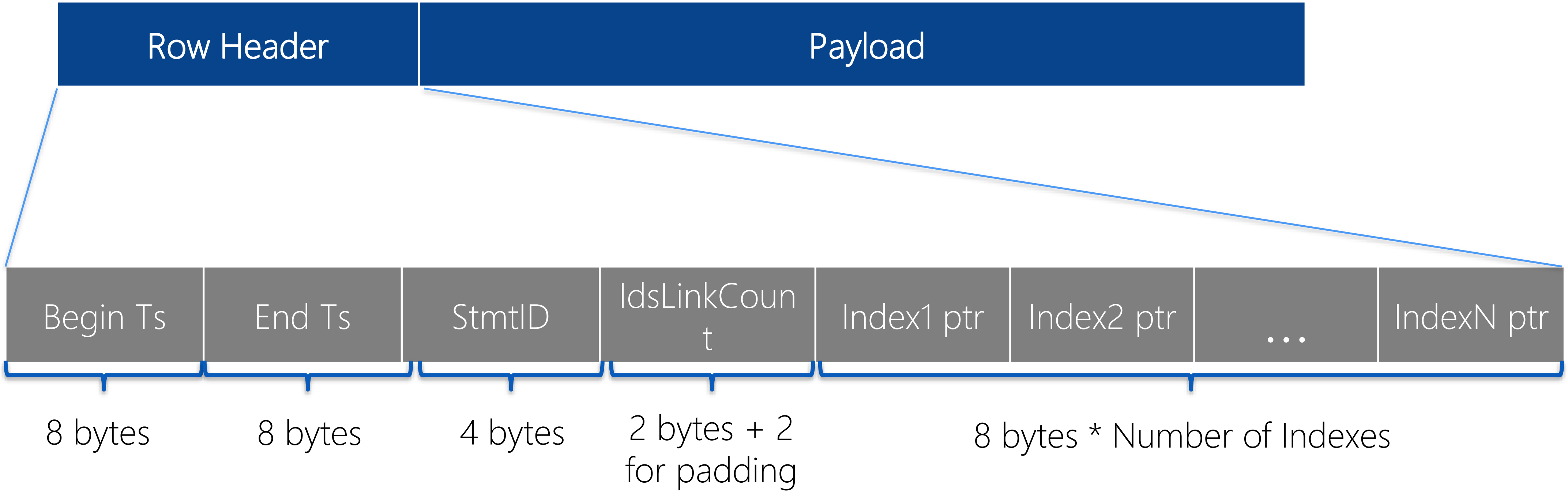
Rows

- Row structure is optimized for memory access
- There are no Pages
- Rows are versioned and there are no in-place updates

Indexes

- There is no clustered index, only non-clustered indexes
- Indexes point to rows, access to rows is via an index
- Indexes do not exist on disk, only in memory, recreated during recovery
- Hash indexes for point lookups
- Range indexes for ordered scans and range scans

In-Memory Row Format



- Begin/End timestamp determines row's version validity and visibility
- No concept of data pages, only rows exist
- Row size limited to 8060 bytes (@table create time) to allow data to be moved to disk-based table
- Not every SQL table schema is supported (Ex: LOB and SqlVariant)

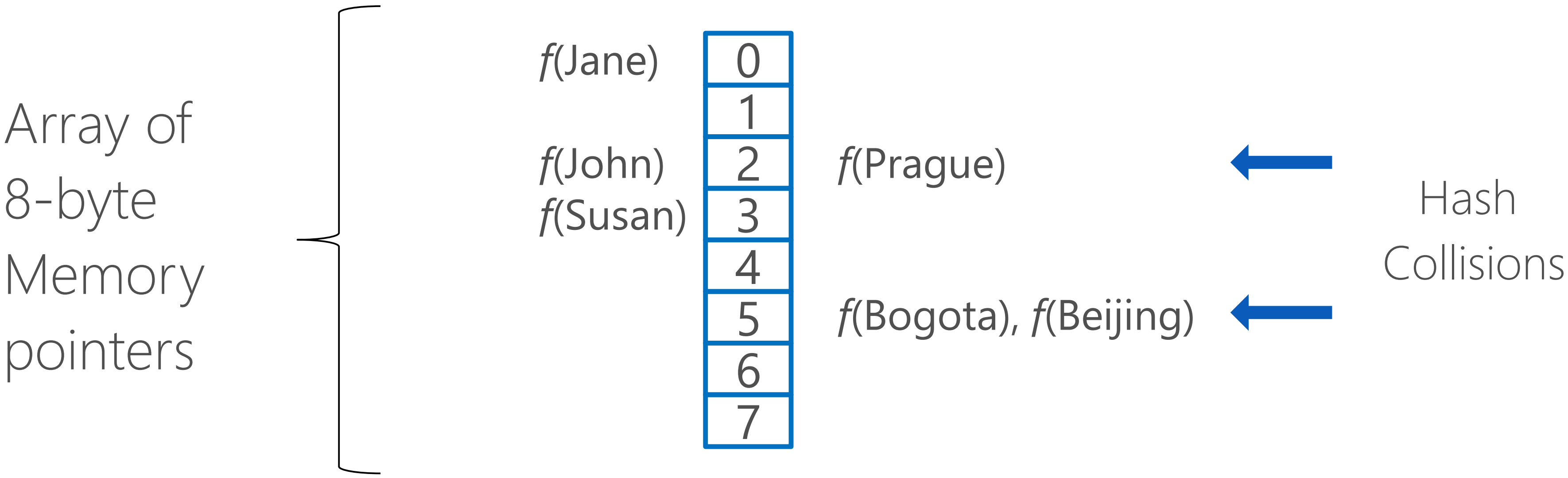
Hash Indexes

Hash index with (bucket_count=8):

Hash function *f*:

- Maps values to buckets
- Built into the system

Hash mapping:



Array of 8-byte Memory pointers

f(Jane) 0

1

f(John) 2 *f*(Prague)

f(Susan) 3

4

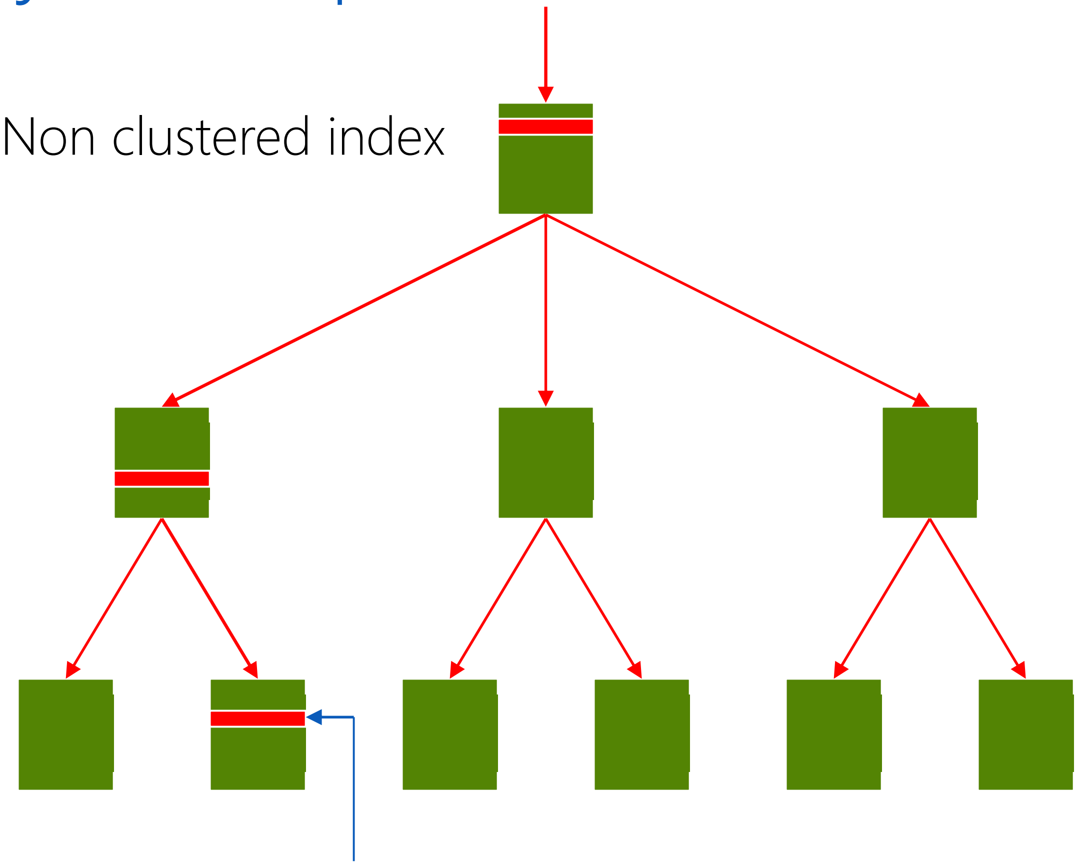
5 *f*(Bogota), *f*(Beijing)

6

7

Hash Collisions

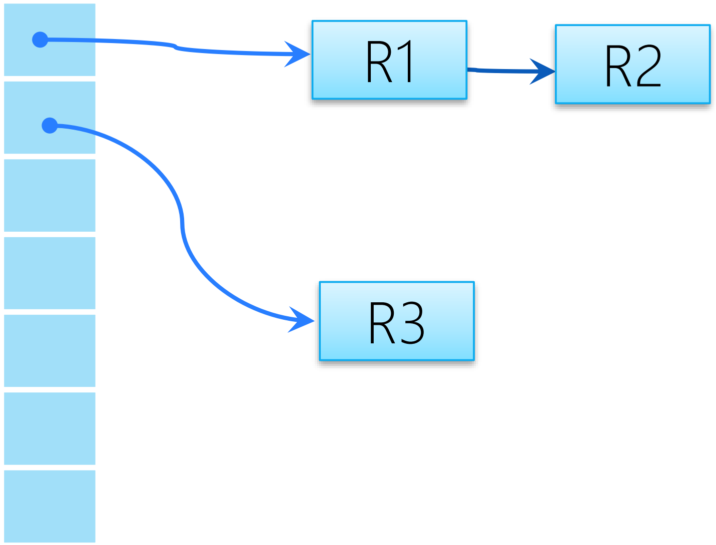
Key Lookup B-Tree vs. Hash Index



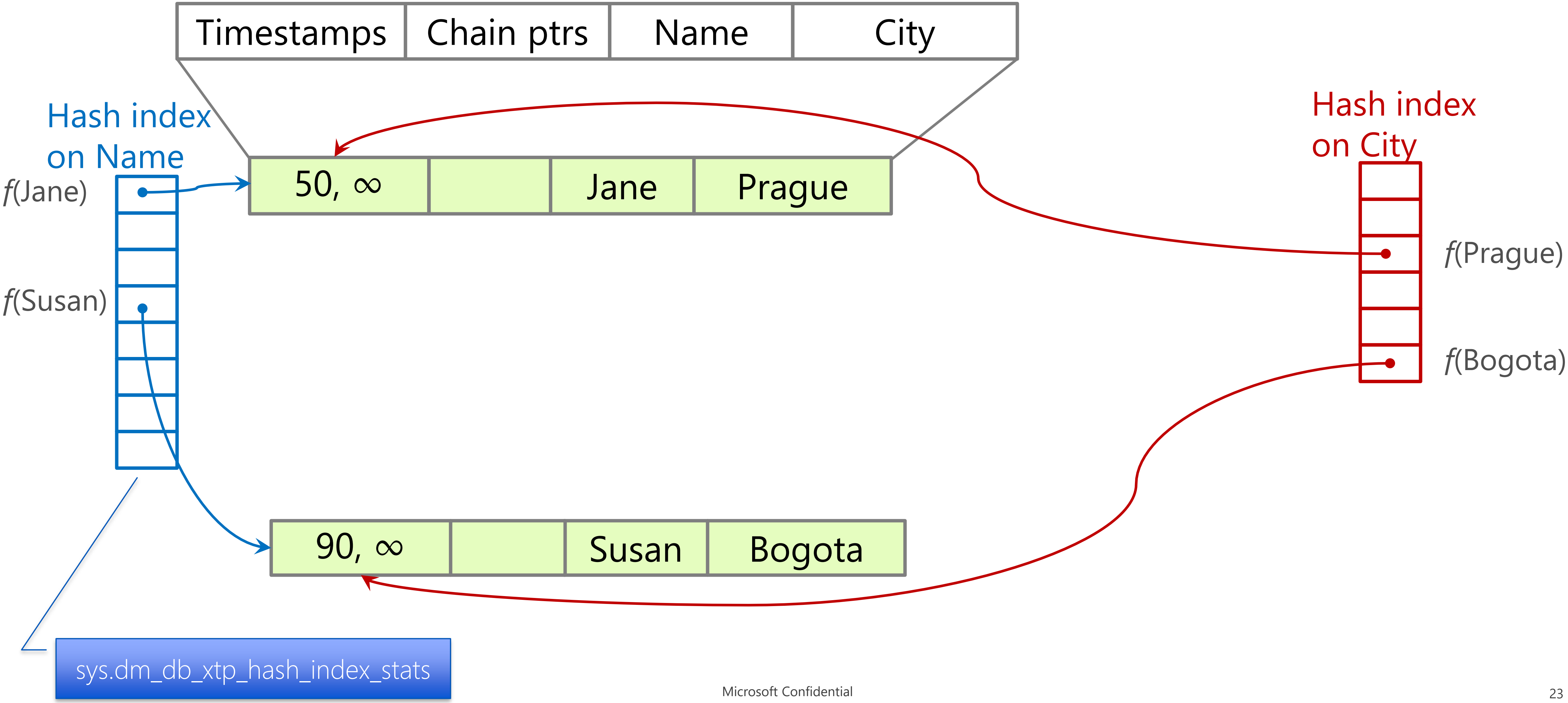
Non clustered index

Matching index record

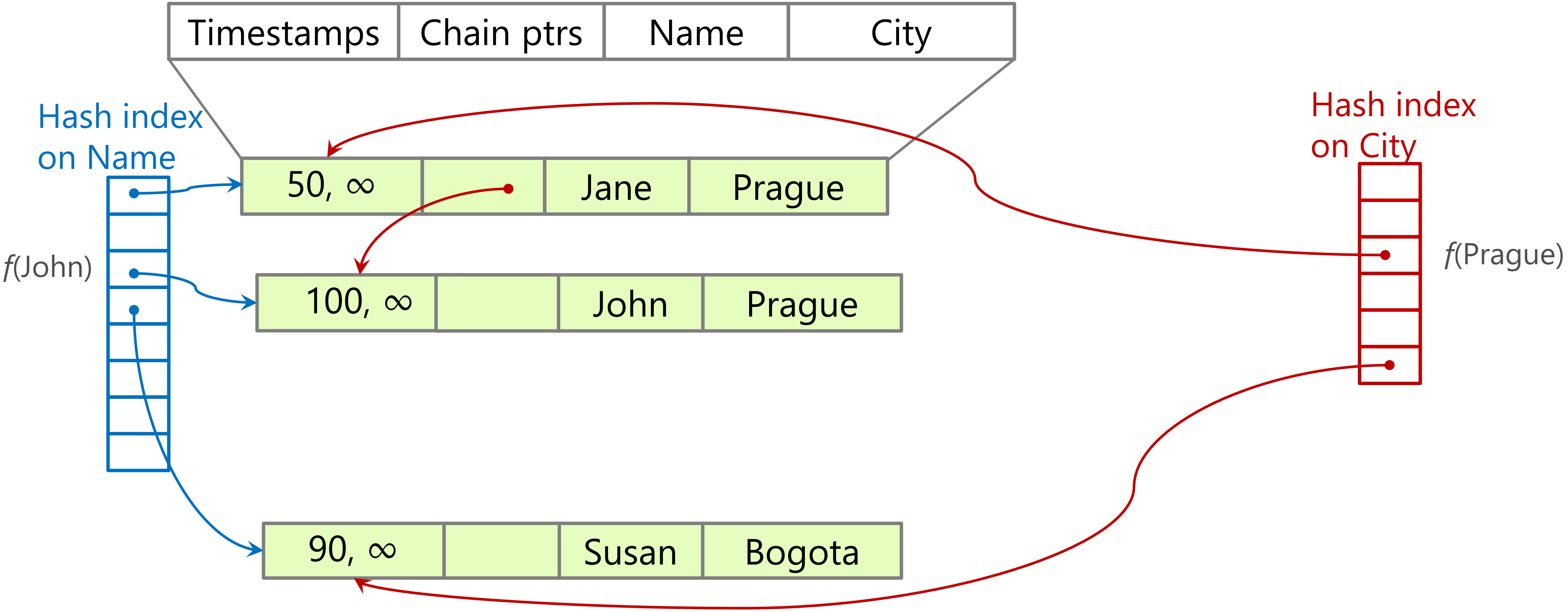
Hash index on Name



Hash Index Traversal

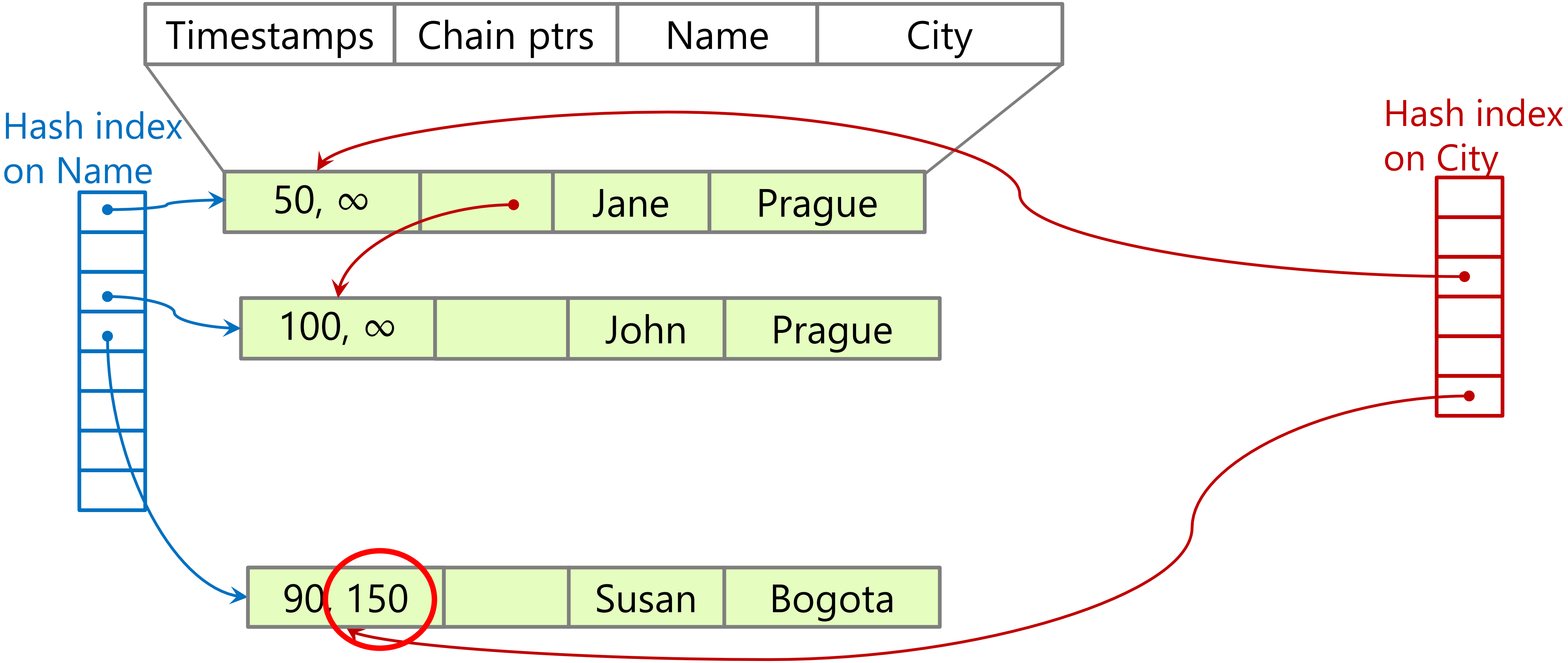


Memory Optimized Table Insert



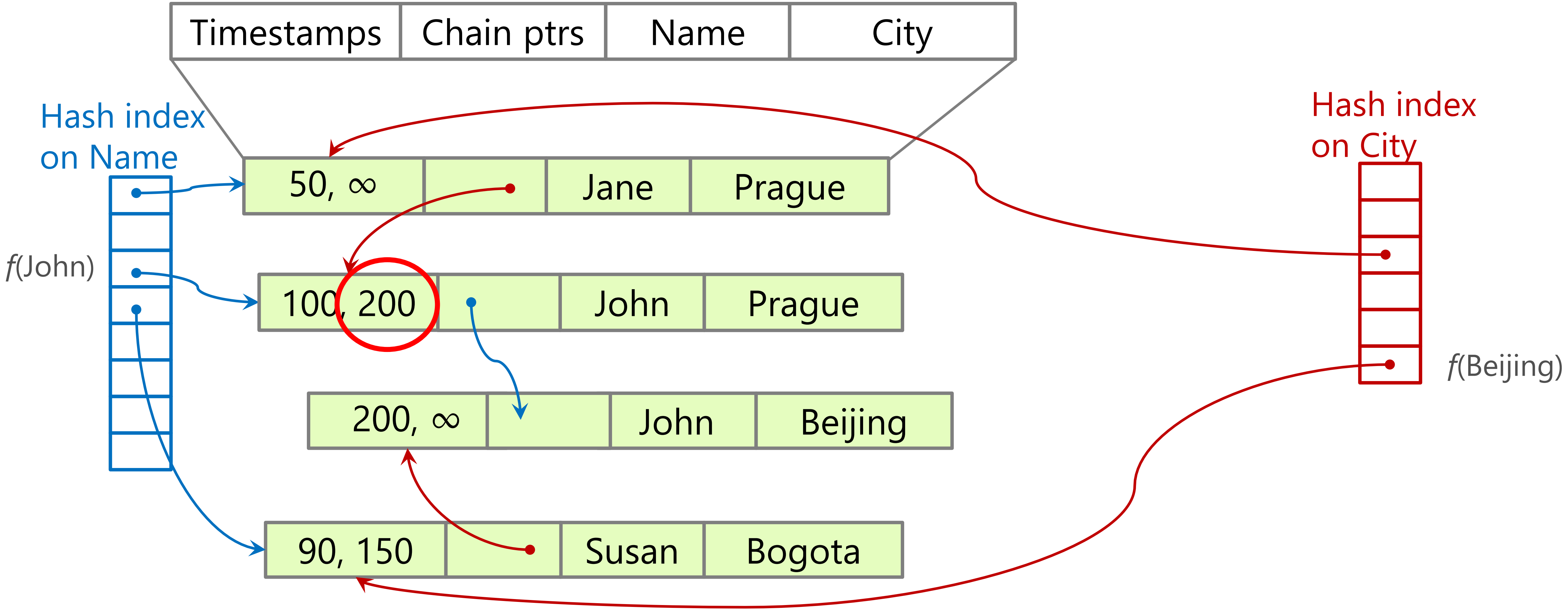
T100: INSERT (John, Prague)

Memory Optimized Table Delete



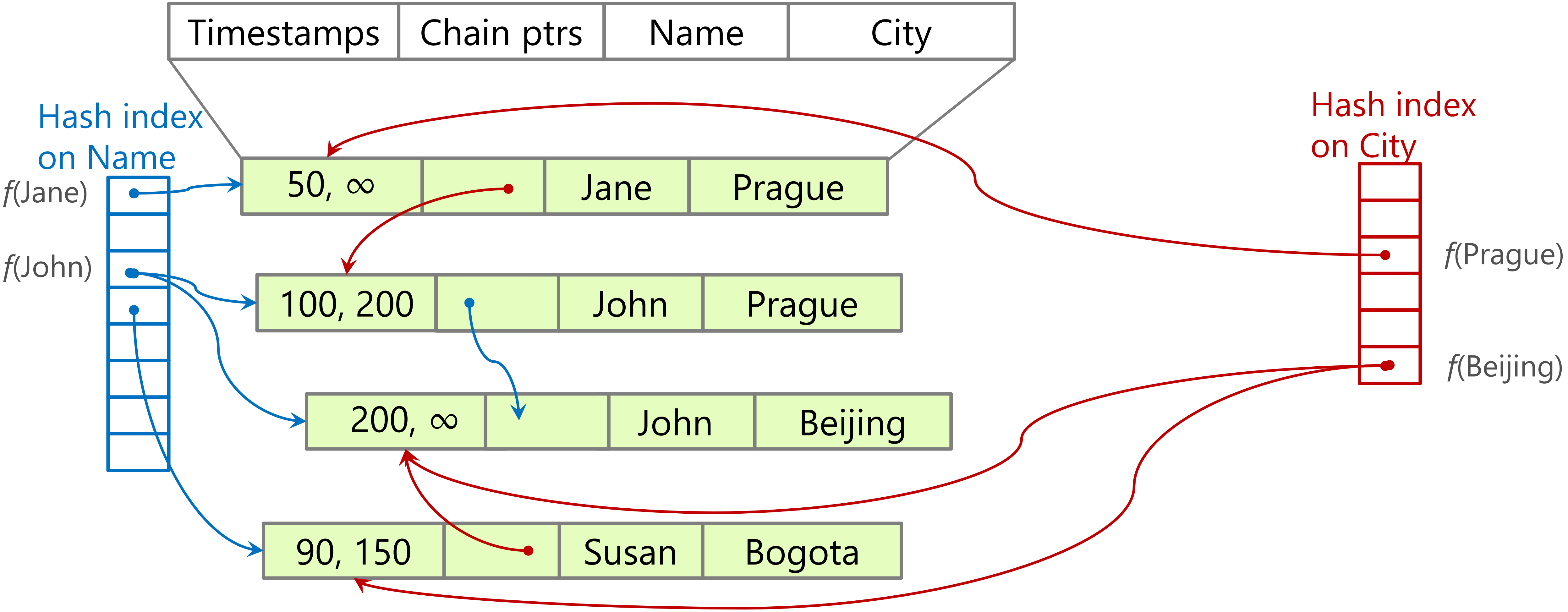
T150: DELETE (Susan, Bogota)

Memory Optimized Table Update



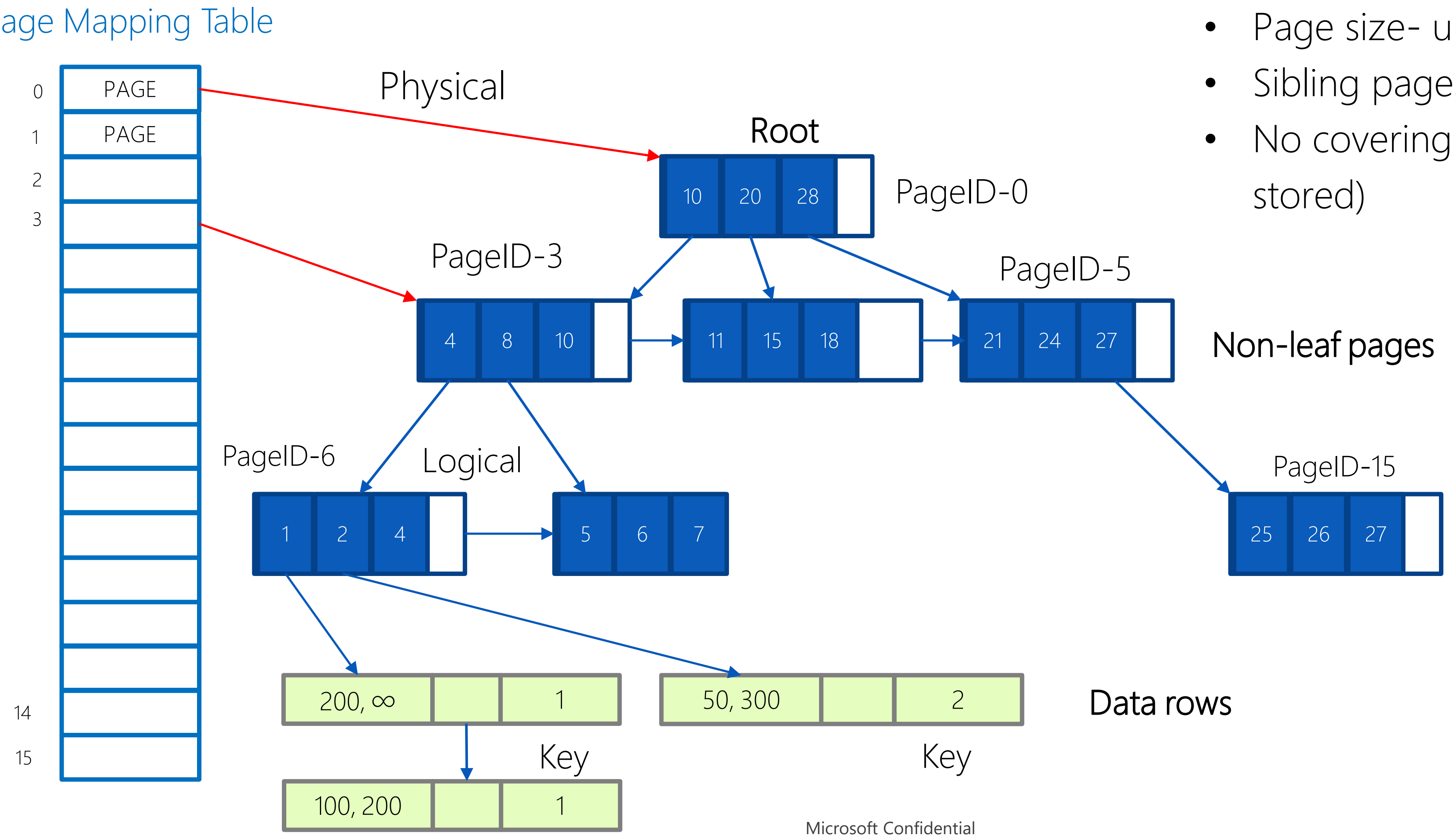
T200: UPDATE (John, Prague) to (John, Beijing)

Garbage Collection



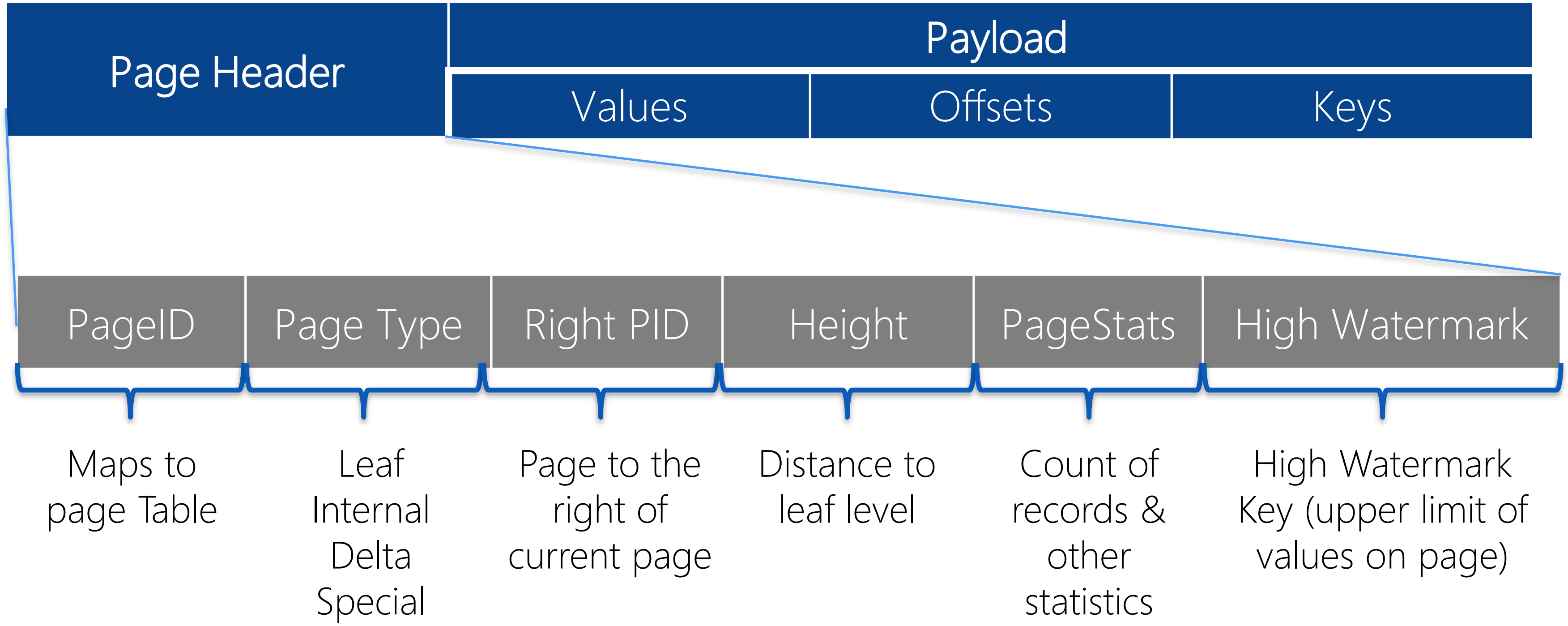
T250: ***Garbage collection***

Non-Clustered (Range) Index



- No latch for page updates
- No in-place updates on index pages
- Page size- up to 8K. Sized to the row
- Sibling pages linked one direction
- No covering columns (only the key is stored)

Range Index Page Format



Maps to page Table

Leaf
Internal
Delta
Special

Page to the right of current page

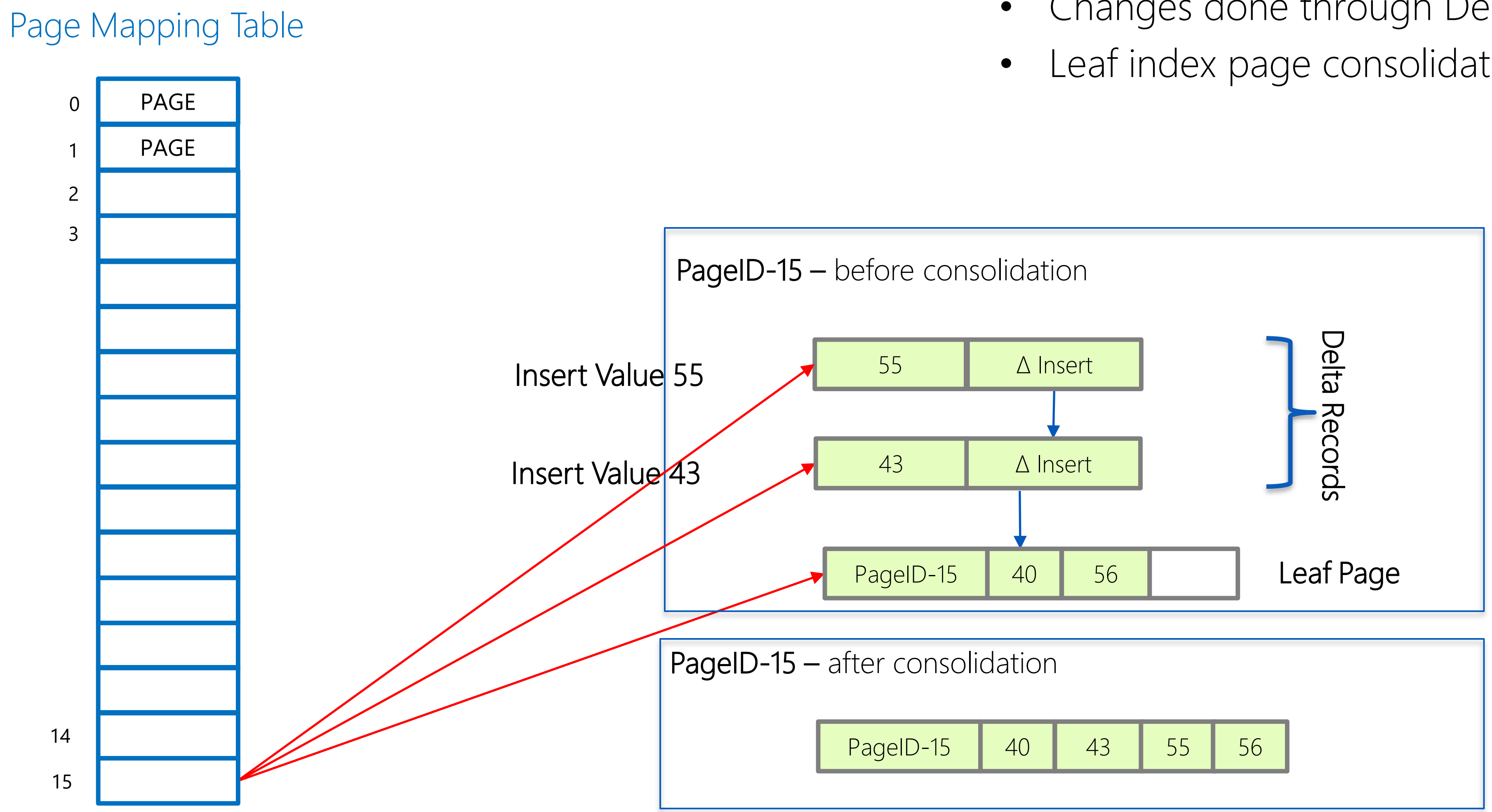
Distance to leaf level

Count of records & other statistics

High Watermark Key (upper limit of values on page)

Non Clustered Index – Delta Consolidation

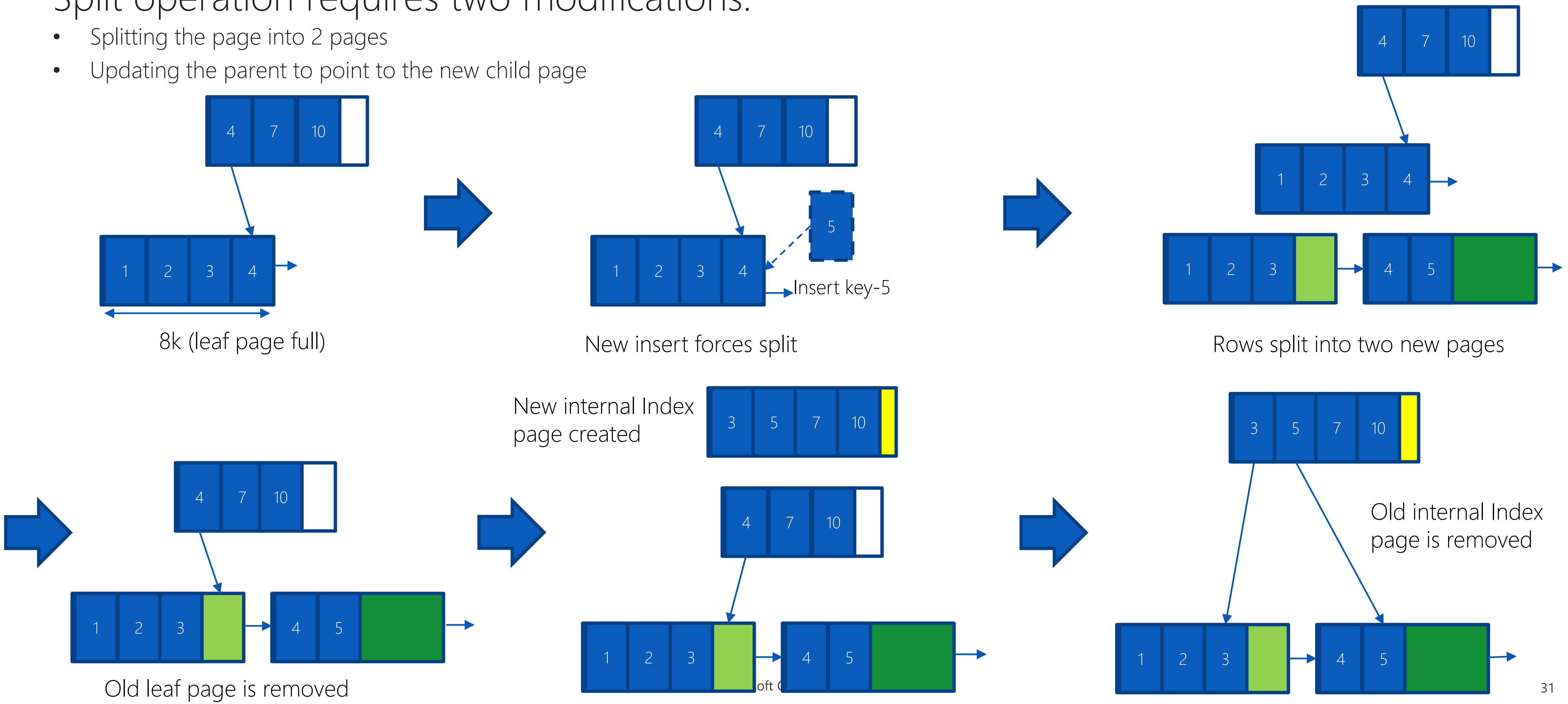
- Changes done through Delta records
- Leaf index page consolidated after 16 delta changes.



Non Clustered Index – Split Operation

Split operation requires two modifications:

- Splitting the page into 2 pages
- Updating the parent to point to the new child page



8k (leaf page full)

Insert key-5

New insert forces split

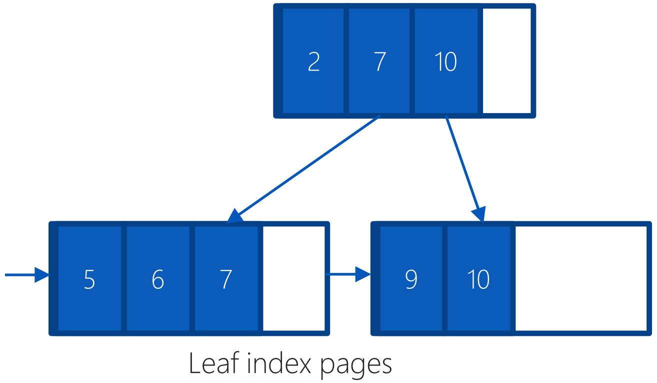
Rows split into two new pages

New internal Index page created

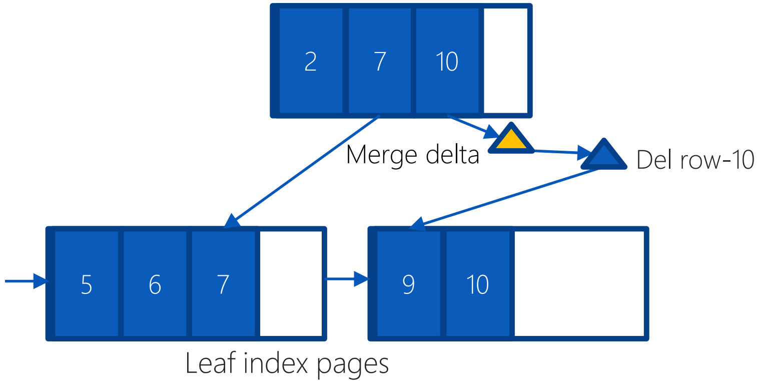
Old leaf page is removed

Old internal Index page is removed

Range Index – Merge Operation

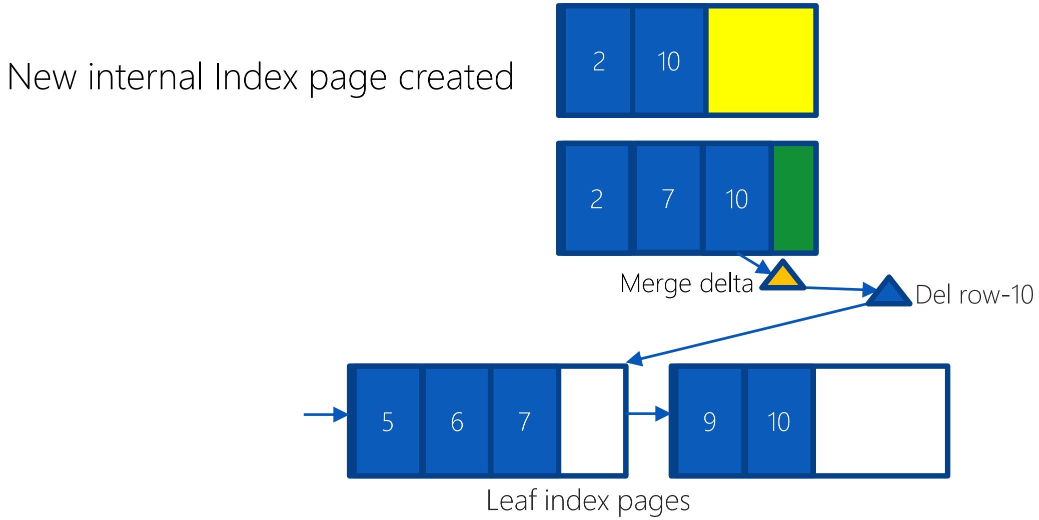


Operation - Delete row 10



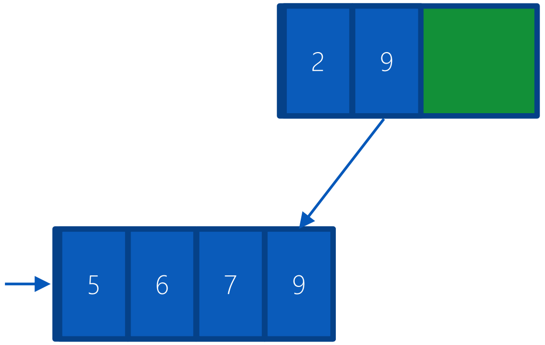
Row 10 marked deleted

STEP – 1
(Insert delta page for merge and for row-10)



Row 10 marked deleted

STEP – 2
(Remove the parent row with Key-7 (requires new index page) and point key-10 to left leaf page)



Leaf index pages merged into a new leaf page

STEP – 3
(Remove the delta pages and move key-9 to the left page)

Index Choice Considerations

Operation	Hash index	Nonclustered index (Range)	Disk-based index
Index Scan	Yes	Yes	Yes
Index seek on equality predicate(s) (=)	Yes (Full key required)	Yes (*)	Yes
Index seek on inequality (>, <, <=,’>=)	No (results in an index scan)	Yes (*)	Yes
Sort-order matching the index definition	No	Yes	Yes
Sort-order matching the reverse of the index definition	No	No	Yes

Yes → index can adequately service the request

No → index cannot be used successfully to satisfy the request.

(*) → For a non-clustered memory-optimized index, the full key is not required to perform an index seek

Durability

Memory optimized tables can be durable or non-durable

- Non-durable tables are ideal for transient data
- Default is durable

```
CREATE TABLE [dbo].[SalesOrder_MemOpt] (
[Order_ID] INT NOT NULL,
[Order_Date] DATETIME NOT NULL,
[Amount] FLOAT NOT NULL,
CONSTRAINT PK_SalesOrderID PRIMARY KEY NONCLUSTERED HASH (Order_ID)
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Single memory optimized filegroup

- Sequential IO pattern (no random IO)
- Filegroup can have multiple containers to aid in parallel recovery
- Recovery depends on IO speed – recommend SSD or Fast SAS drives

Storage

Filestream container is the underlying storage mechanism

- Checksums and single-bit correcting ECC on files

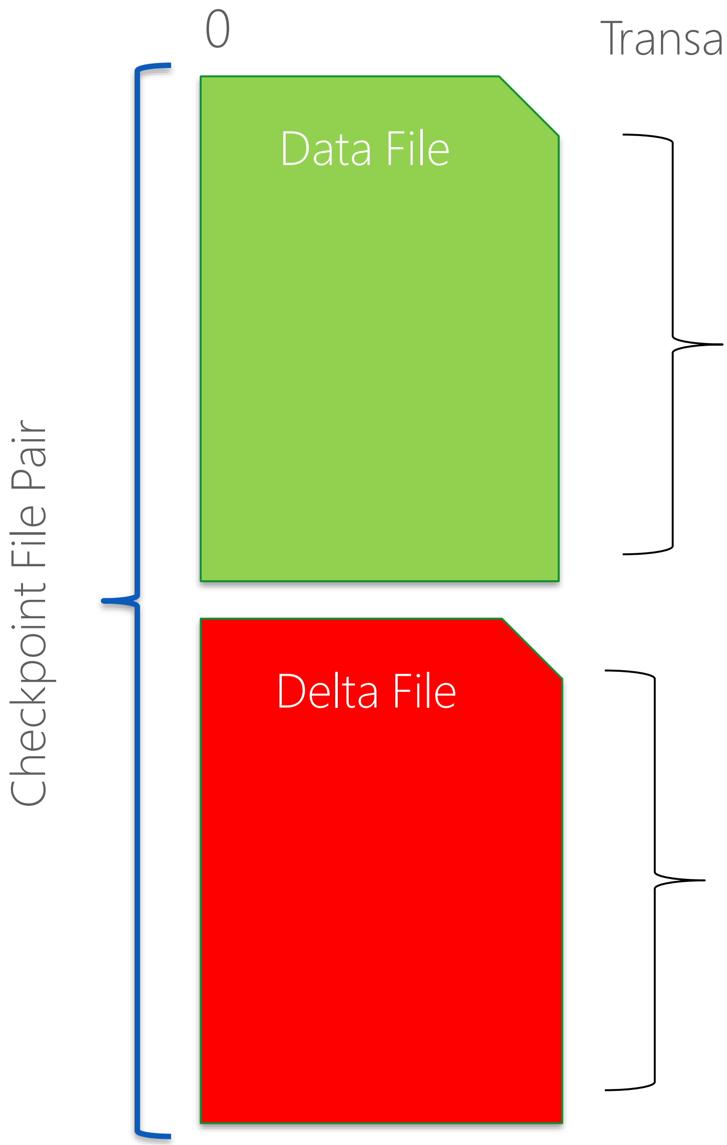
Data files

- ~128MB in size, write 256KB chunks at a time
- Stores only the inserted rows (i.e. table content)
- Chronologically organized streams of row versions

Delta files

- File size is not constant, write 4KB chunks at a time (16MB if Server has > 16GB RAM)
- Stores IDs of deleted rows

Data and Delta Files



Transaction Timestamp Range

TS (ins)	RowId	TableId	Row pay load
TS (ins)	RowId	TableId	Row pay load
TS (ins)	RowId	TableId	Row pay load

Data file contains rows inserted within a given transaction range

TS (ins)	RowId	TS (del)
TS (ins)	RowId	TS (del)
TS (ins)	RowId	TS (del)

Delta file contains deleted rows within a given transaction range

Checkpoint Process

Not tied to a “recovery interval”

“Offline Checkpoint” log records and flushes data to the data and delta files in the filestream container

Invoked in multiple ways:

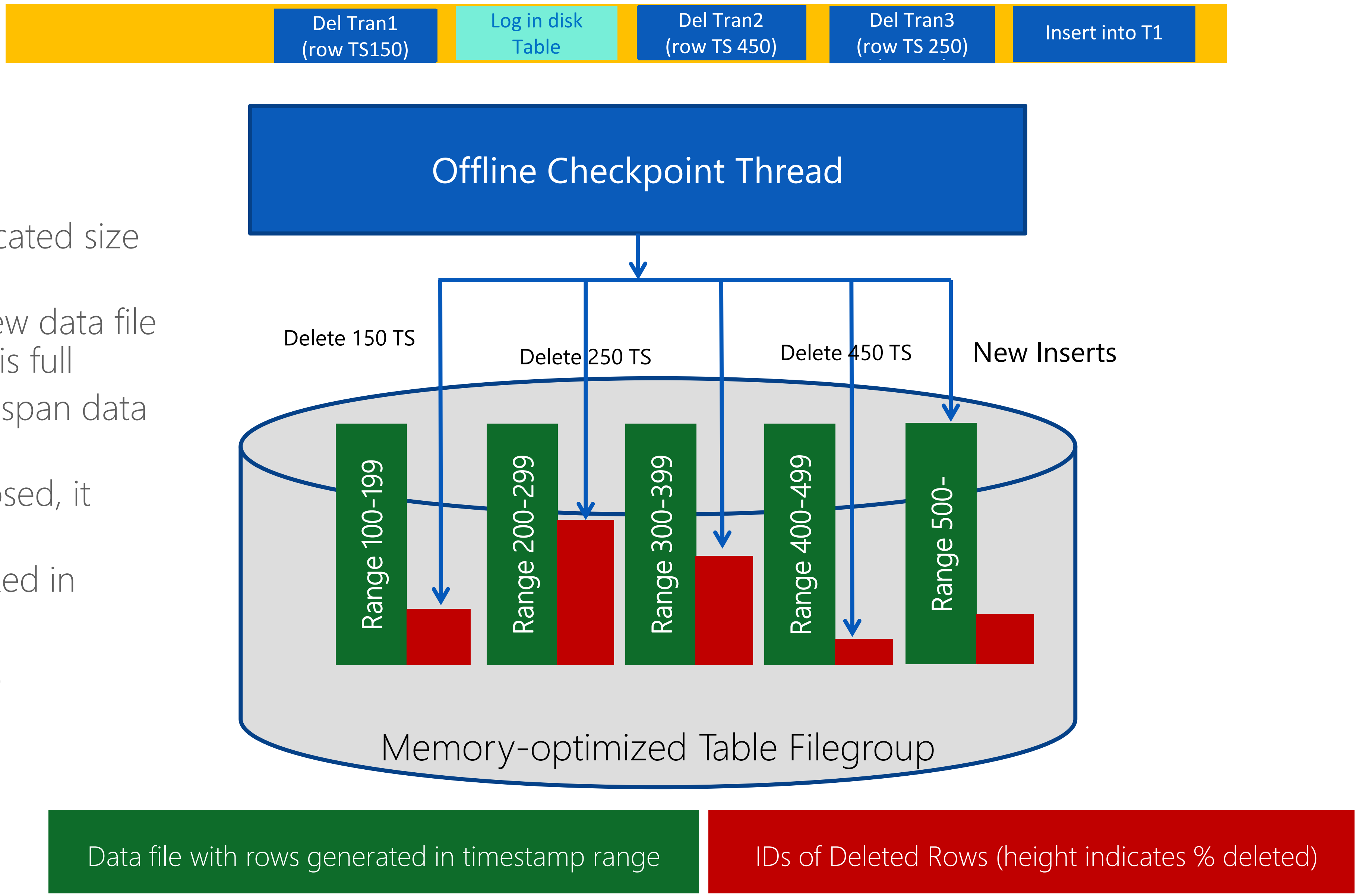
- Manual Checkpoint – Explicit checkpoint command issued
- Size based – Log grown by 512MB
- Time Based – if the time since last checkpoint issued exceeds threshold

SE_MANAGE_VOLUME recommended for performance reasons (instant file initialization)

Troubleshooting: sys.dm_db_xtp_checkpoint_stats

Populating Data and Delta Files

SQL Transaction log (LogPool)



- Data file has pre-allocated size (128 MB)
- Engine switches to new data file when the current file is full
- Transaction does not span data files
- Once a data file is closed, it becomes read-only
- Row deletes are tracked in delta file
- Files are append only

Merge Operation

What is a Merge Operation?

- Merges two or more adjacent data/delta files pairs into 1 pair

Need for Merge

- Deleting rows causes data files to have stale rows
- DMV: *sys.dm_xtp_checkpoint_files* can be used to find inserted/deleted rows and free space

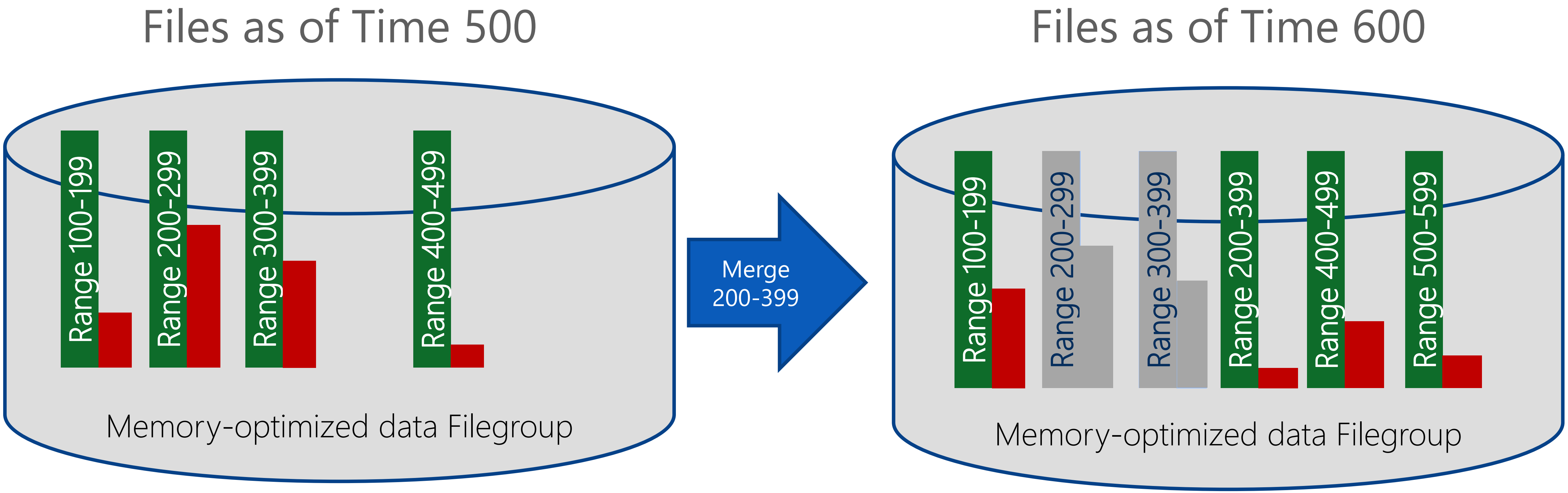
Benefits of Merge

- Reduces storage (i.e. fewer data/delta files) required to store active data rows
- Improves the recovery time as there will be fewer files to load

When is Merge done?

- 2 or more consecutive CFPs can be consolidated, after accounting for deleted rows, such that the resultant rows can fit into 1 CFP of ideal size
- single CFP can be self-merged the data file exceeds 256 MB and over half of the rows are deleted
- Manual Merge invoked calling *sys.sp_xtp_merge_checkpoint_files*

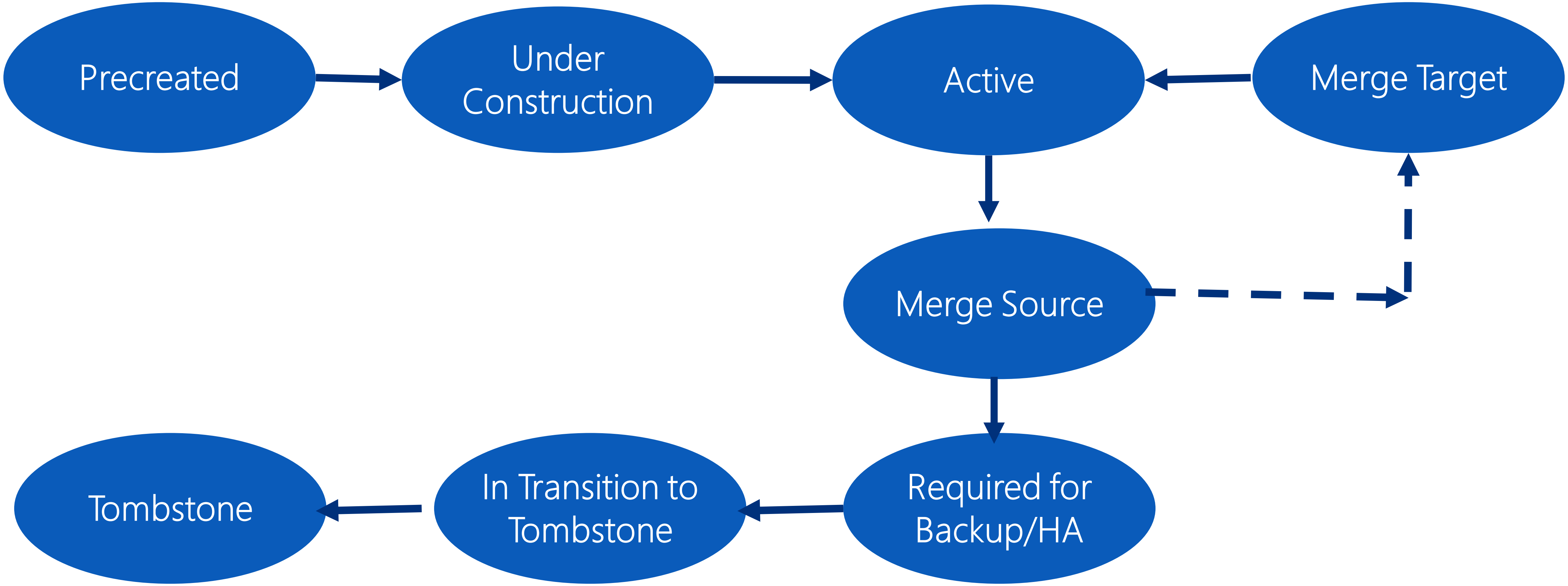
Merge



Data file with rows generated in timestamp range	IDs of Deleted Rows (height indicates % deleted)	Files Under Merge	Deleted Files

Checkpoint Files States

sys.dm_db_xtp_checkpoint_files
Column → state_desc



Logging

Uses SQL transaction log to store content

- Each In-Memory OLTP log record contains a log record header followed by memory optimized-specific log content

All logging for memory-optimized tables is logical

- No log records for physical structure modifications
- No index-specific / index-maintenance log records
- Log records are written only on a commit, no UNDO information is logged

Latency

- Latency is absolutely critical, recommend SSDs

Recovery Models

- All three recovery models are supported

Delayed Durability

Transaction commits logged asynchronously

Set at Database Level or Atomic Block

- Disabled – Normal behavior durability guaranteed.
- Allowed – Allowed at the DB Level, Transaction has to specify durability options, default is a durable transaction.

  COMMIT TRAN……. WITH (DELAYED_DURABILITY=ON)
- FORCED – Changes default durability for the DB to “delayed”. Can be useful for applications bottlenecked on Log IO, that can tolerate some data loss on a failure.

```
CREATE PROCEDURE MyProc
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS BEGIN ATOMIC WITH
(DELAYED_DURABILITY = ON,
TRANSACTION ISOLATION LEVEL = SNAPSHOT,
LANGUAGE = N'us_english')
        -- Insert T-SQL here…
END
```

```
ALTER DATABASE Hekaton
SET DELAYED_DURABILITY = FORCED
GO
```

Delayed Durability

Applicable for both memory-optimized and normal tables

Durability automatically managed by System behind the scenes

Transaction flush guaranteed if:

- Durable transaction is executed
- Manually execute sp_flush_log

All System transactions are durable

Note: Data loss possibility if the server goes down between the time a transaction is committed and its log record if flushed

Diagnostics

Log not being truncated

- sys.databases log_reuse_wait_desc='XTP_CHECKPOINT'.

DMV's / DMF's

- sys.dm_db_xtp_checkpoint_stats
- sys.dm_db_xtp_checkpoint_files
- sys.fn_dblog_xtp()/sys.fn_dump_log_xtp()

Manual merge

- sys.sp_merge_xtp_checkpoint_files

Database Recovery – Memory Optimized Tables

Analysis Phase

- Finds the last completed checkpoint

Data Load

- Instantiate memory optimized tables
- Load from set of data/delta files from the last completed checkpoint
- Parallel Load by reading data/delta files using 1 thread / file

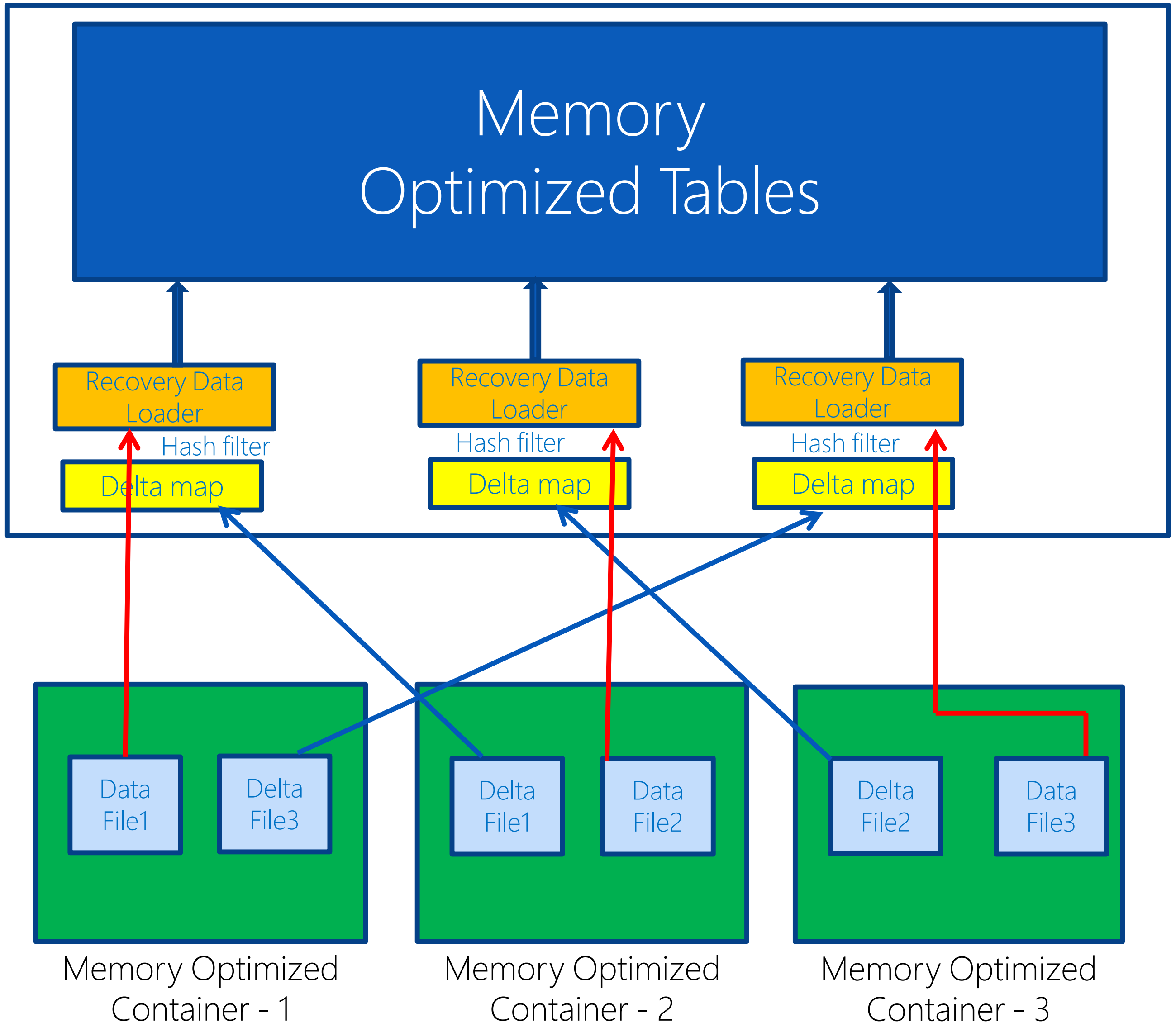
Redo phase

- Apply the transaction log from last checkpoint
- Concurrent with REDO on disk-based tables

No UNDO phase

- Since only committed transactions are logged

Parallel Data Load



Impact on RTO

- Load speed of data
- Size of durable tables

Limit stands at 8192 data/delta file pairs per database

- If 8192 data files were fully utilized at 128MB a piece, it would be able to hold almost 1TB of data
- We recommend durable table database size to be 250GB or less

Microsoft Confidential

Recovery Progress and Diagnostics

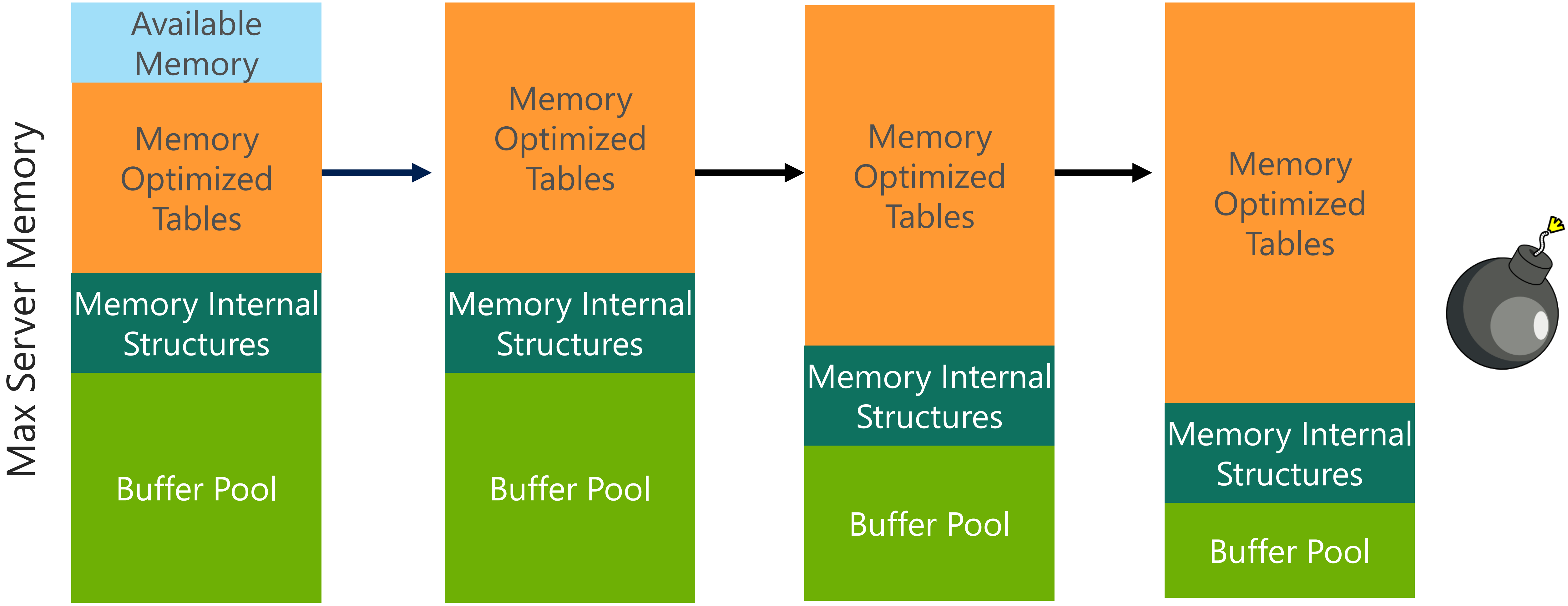
Progress

- Extended Event xtp_recover_table
- Extended Event xtp_recover_done
- Errorlog

Recovery Failures

- Due to compilation
- Due to memory availability/configured to load the data
- Checksum failure on checkpoint files
- Table size limitation (checkpoint files limit is 8192)
- Other normal recovery failure reasons

Memory problem?



Max Server Memory

Available Memory

Memory Optimized Tables

Memory Internal Structures

Buffer Pool

Memory Optimized Tables

Memory Internal Structures

Buffer Pool

Memory Optimized Tables

Memory Internal Structures

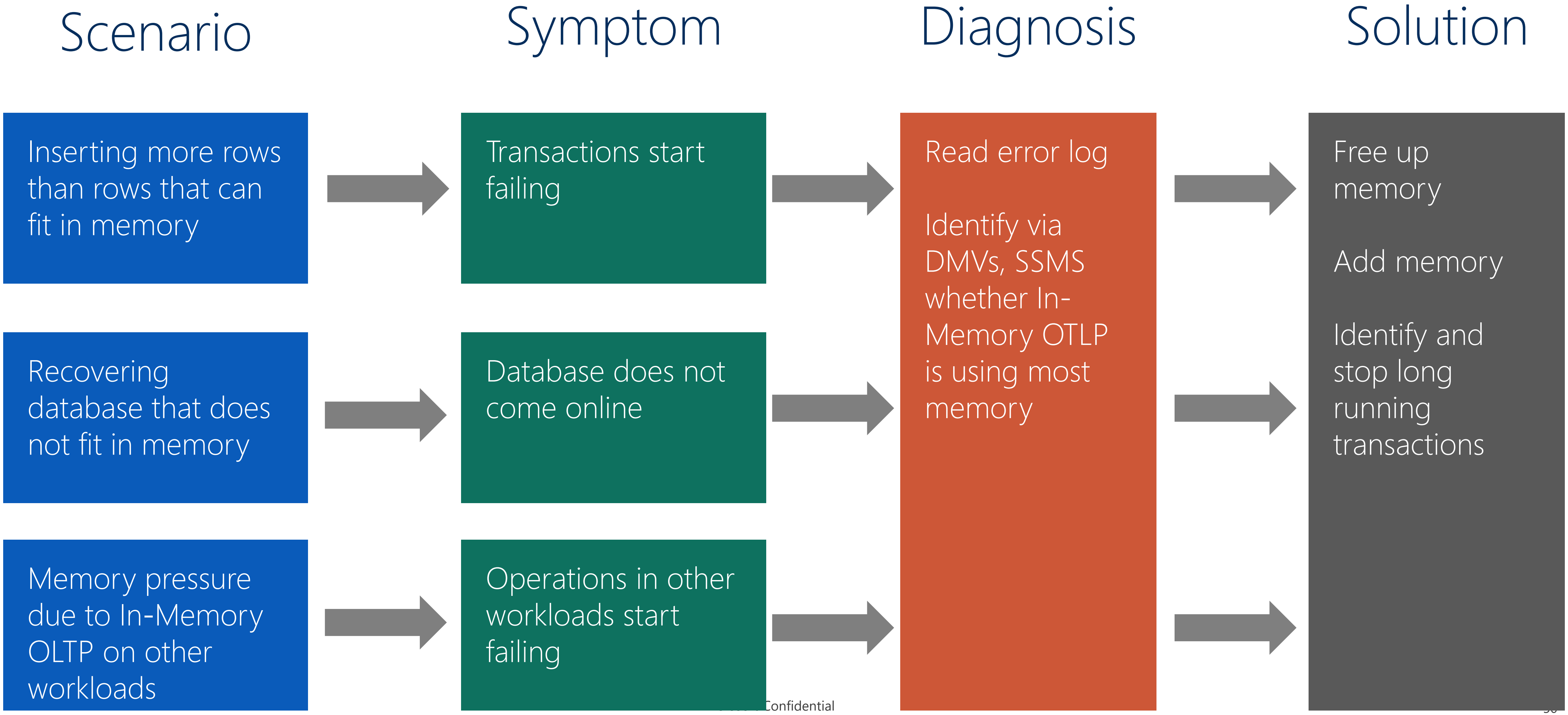
Buffer Pool

Memory Optimized Tables

Memory Internal Structures

Buffer Pool

Memory Considerations



Scenario

Inserting more rows than rows that can fit in memory

Recovering database that does not fit in memory

Memory pressure due to In-Memory OLTP on other workloads

Symptom

Transactions start failing

Database does not come online

Operations in other workloads start failing

Diagnosis

Read error log

Identify via DMVs, SSMS whether In-Memory OTLP is using most memory

Solution

Free up memory

Add memory

Identify and stop long running transactions

Memory Considerations

Data resides in memory at all times

- Configure SQL server with sufficient memory to store memory-optimized tables
- Failure to allocate memory will fail transactional workload at run-time
- Other SQL workloads can slow down to unacceptable performance

Sizing

- Table Size: rule of thumb 2* size of data
- Hash Indexes: 8KB * [actual bucket count]
- Non clustered index: variable size

Management

- Limit memory consumption using Resource Governor
- DMVs, Perfmon
- Freeing memory is not synchronous in most cases

Resource Governor Integration

Limits max memory allocation for In-Memory OLTP

Prevents performance degradation of regular SQL workloads

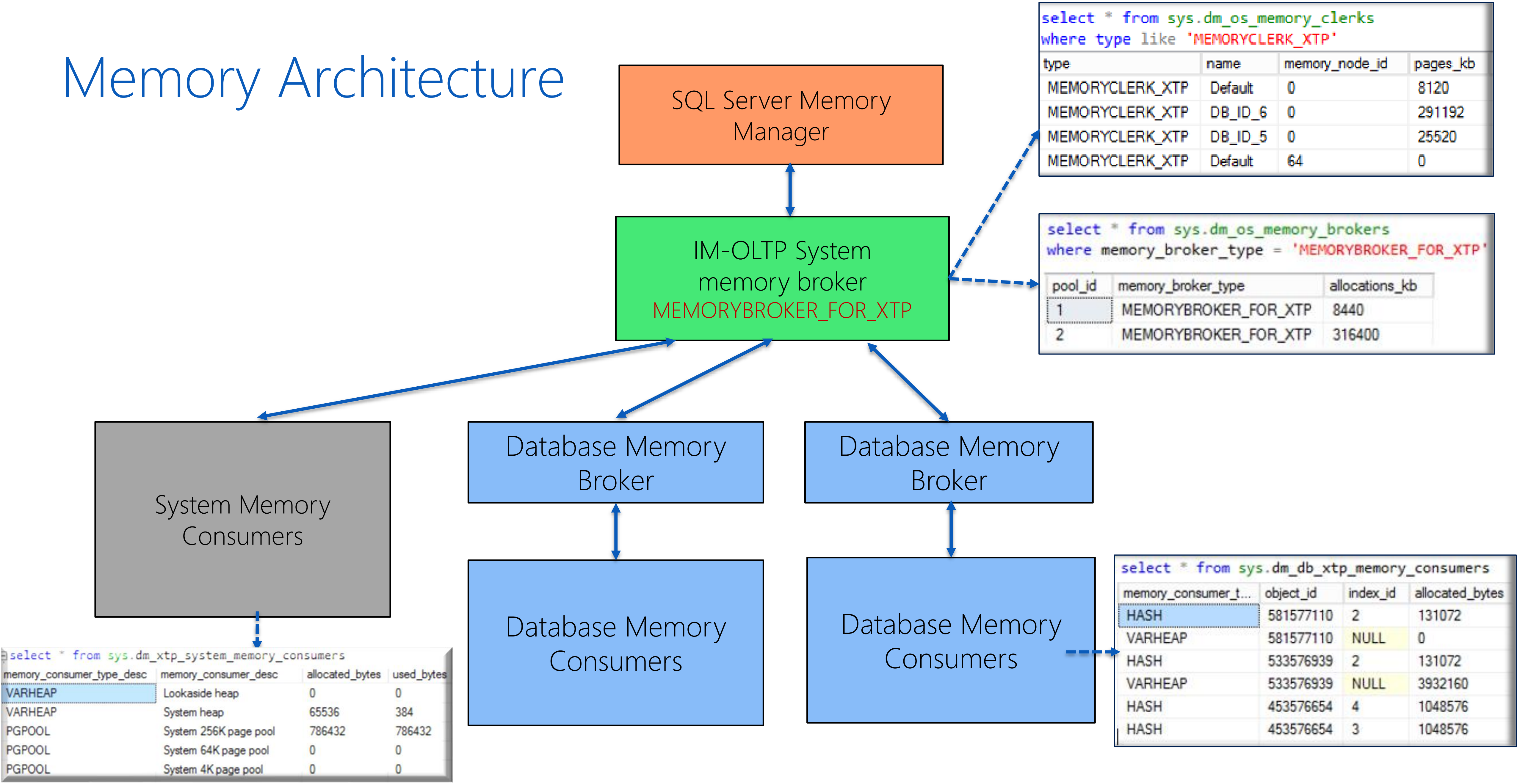
Implementation:

- Define dedicated pool for In-Memory OLTP
- Bind DB to resource pool defined

  → *sys.sp_xtp_bind_db_resource_pool*
- Set database offline and Online

Internal pool target memory	%available of dedicated pool before OOM notification
<= 8 GB	70%
<= 16GB	75%
<= 32GB	80%
<= 96GB	85%
>= 96 GB	90%

Memory Architecture



SQL Server Memory Manager

IM-OLTP System memory broker MEMORYBROKER_FOR_XTP

System Memory Consumers

Database Memory Broker

Database Memory Broker

Database Memory Consumers

Database Memory Consumers

```
select * from sys.dm_os_memory_clerks
where type like 'MEMORYCLERK_XTP'
```

type	name	memory_node_id	pages_kb
MEMORYCLERK_XTP	Default	0	8120
MEMORYCLERK_XTP	DB_ID_6	0	291192
MEMORYCLERK_XTP	DB_ID_5	0	25520
MEMORYCLERK_XTP	Default	64	0

```
select * from sys.dm_os_memory_brokers
where memory_broker_type = 'MEMORYBROKER_FOR_XTP'
```

pool_id	memory_broker_type	allocations_kb
1	MEMORYBROKER_FOR_XTP	8440
2	MEMORYBROKER_FOR_XTP	316400

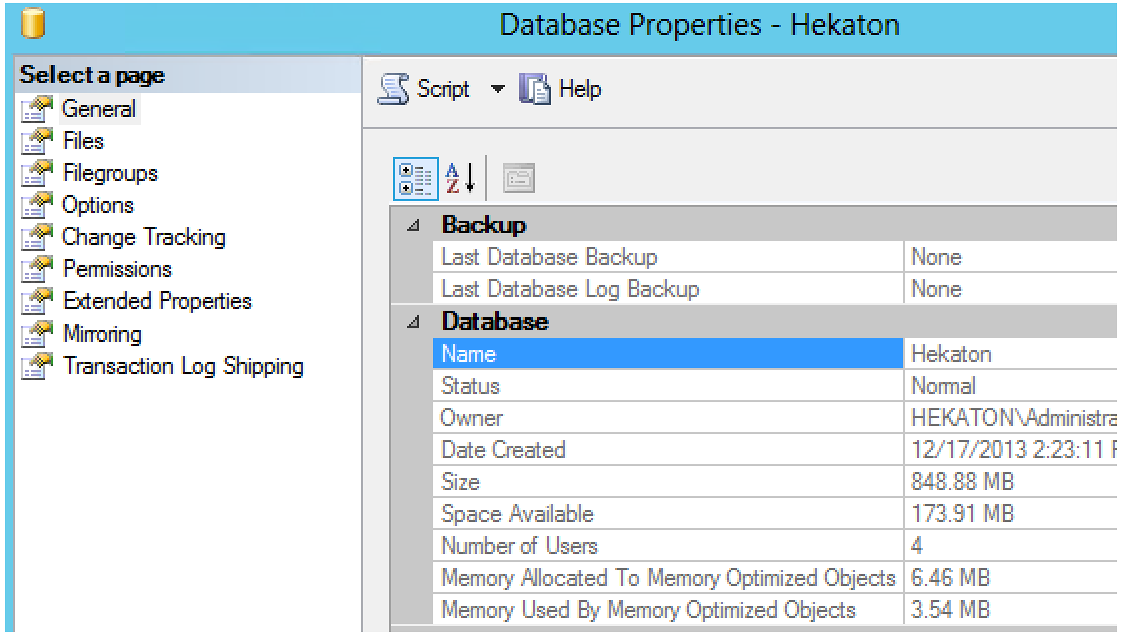
```
select * from sys.dm_xtp_system_memory_consumers
```

memory_consumer_type_desc	memory_consumer_desc	allocated_bytes	used_bytes
VARHEAP	Lookaside heap	0	0
VARHEAP	System heap	65536	384
PGPOOL	System 256K page pool	786432	786432
PGPOOL	System 64K page pool	0	0
PGPOOL	System 4K page pool	0	0

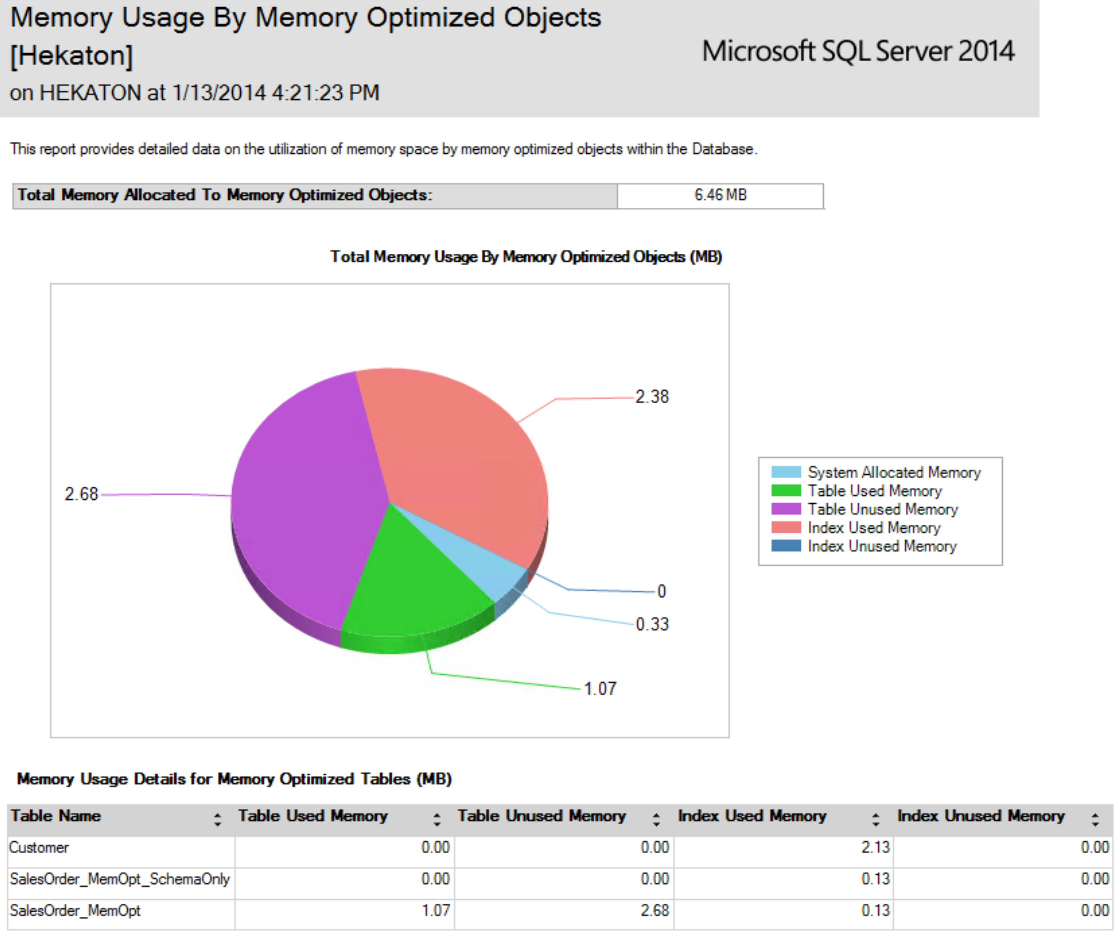
```
select * from sys.dm_db_xtp_memory_consumers
```

memory_consumer_t...	object_id	index_id	allocated_bytes
HASH	581577110	2	131072
VARHEAP	581577110	NULL	0
HASH	533576939	2	131072
VARHEAP	533576939	NULL	3932160
HASH	453576654	4	1048576
HASH	453576654	3	1048576

Monitoring



Backup	
Last Database Backup	None
Last Database Log Backup	None
Database	
Name	Hekaton
Status	Normal
Owner	HEKATON\Administra
Date Created	12/17/2013 2:23:11 F
Size	848.88 MB
Space Available	173.91 MB
Number of Users	4
Memory Allocated To Memory Optimized Objects	6.46 MB
Memory Used By Memory Optimized Objects	3.54 MB



Memory Usage By Memory Optimized Objects
[Hekaton]
on HEKATON at 1/13/2014 4:21:23 PM

Microsoft SQL Server 2014

This report provides detailed data on the utilization of memory space by memory optimized objects within the Database.

Total Memory Allocated To Memory Optimized Objects:	6.46 MB

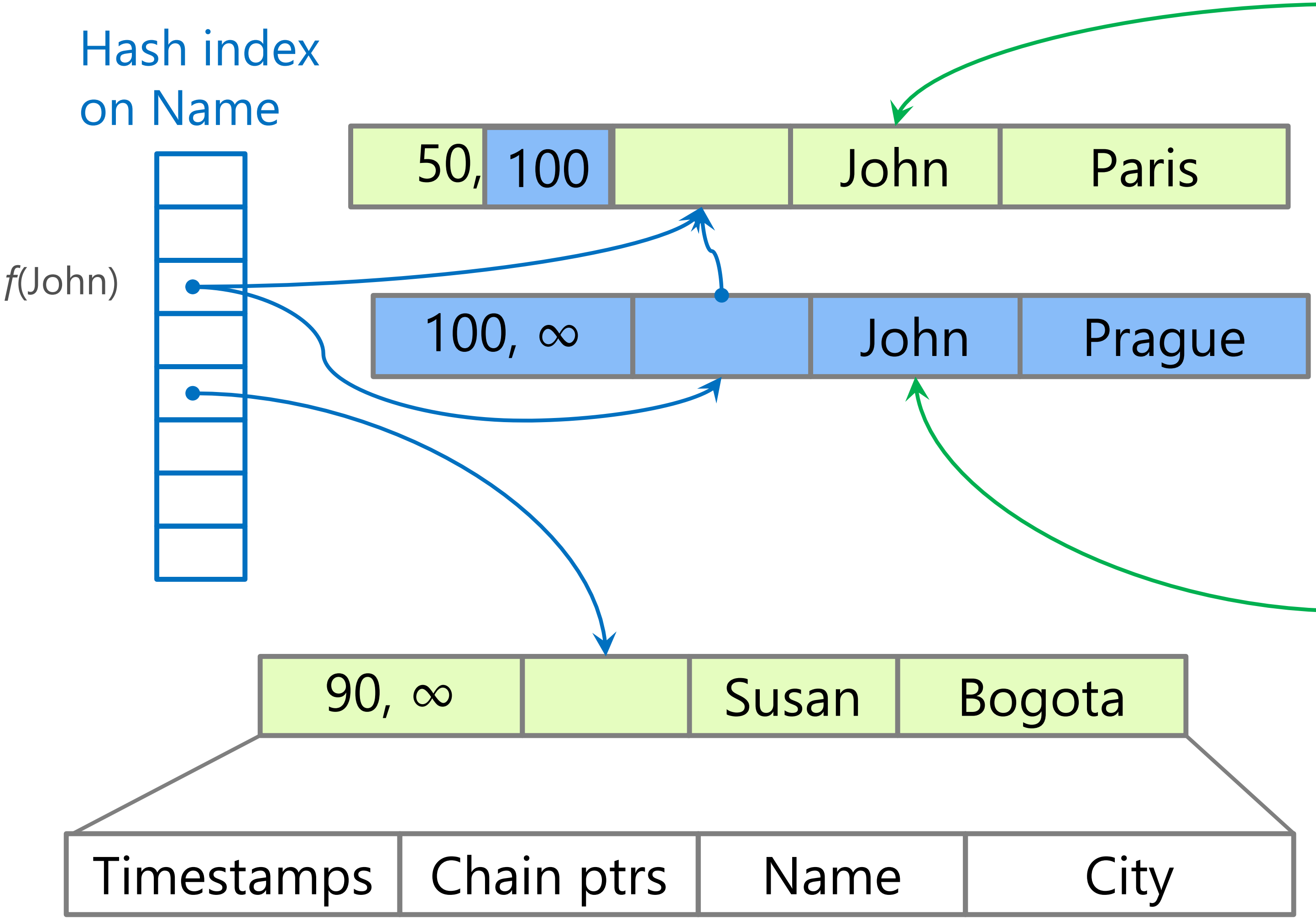
Total Memory Usage By Memory Optimized Objects (MB)

Memory Usage Details for Memory Optimized Tables (MB)

Table Name	Table Used Memory	Table Unused Memory	Index Used Memory	Index Unused Memory
Customer	0.00	0.00	2.13	0.00
SalesOrder_MemOpt_SchemaOnly	0.00	0.00	0.13	0.00
SalesOrder_MemOpt	1.07	2.68	0.13	0.00

Memory Usage	DMV
Memory Optimized Table size	sys.dm_db_xtp_table_memory_stats
Database IM-OLTP memory usage	sys.dm_db_xtp_memory_consumers
Non-Database IM-OLTP memory usage	sys.dm_xtp_system_memory_consumers
Overall IM-OLTP memory usage	sys.dm_os_memory_brokers sys.dm_os_memory_clerks

Multi-Version Lock Free Transactions



Transaction 99: Running compiled query

SELECT City WHERE Name = 'John'

Simple hash lookup returns direct pointer to 'John' row

Background operation will unlink and deallocate the old 'John' row after transaction 99 completes.

Transaction 100:

UPDATE City = 'Prague' where Name = 'John'

No locks of any kind, no interference with transaction 99

Garbage Collection

Stale Row Versions

- Updates, deletes, and aborted insert operations create row versions that (eventually) are no longer visible to any transaction
- Slow down scans of index structures
- Create unused memory that needs to be reclaimed (i.e. Garbage Collected)

Garbage Collection (GC)

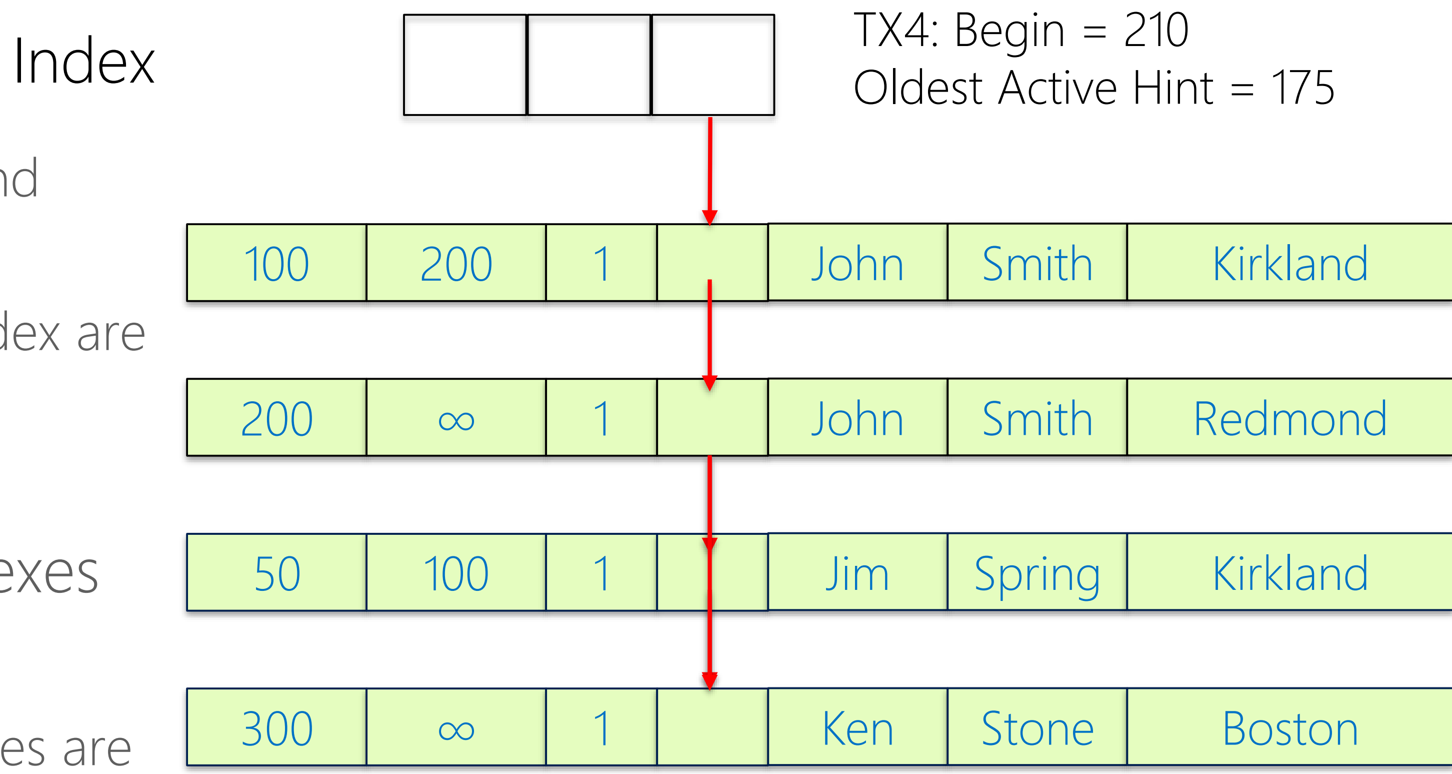
- Analogous to version store cleanup task for disk-based tables to support Read Committed Snapshot (RCSI)
- System maintains ‘oldest active transaction’ hint

Cooperative Garbage Collection

- Scanners can remove expired rows when found
- Offloads work from GC thread
- Ensures that frequently visited areas of the index are cleaned regularly

A row needs to be removed from all indexes before memory can be freed

- Garbage collection is most efficient if all indexes are frequently accesses



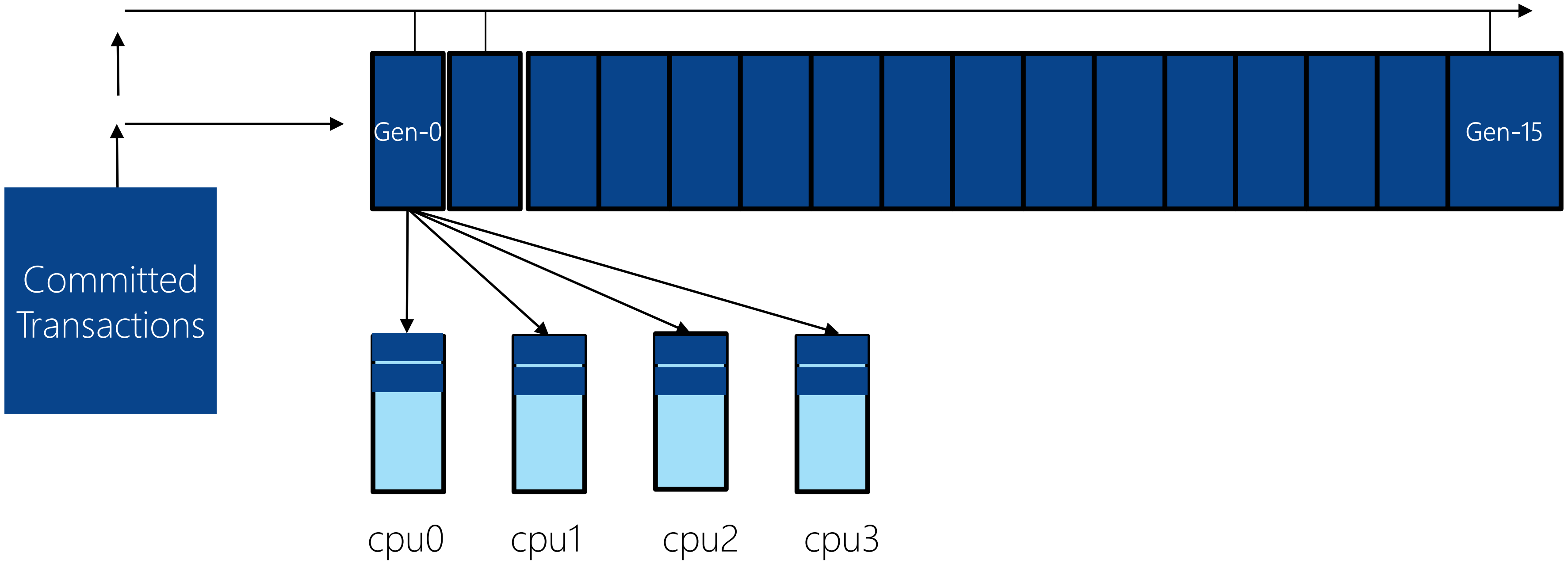
Index

TX4: Begin = 210
Oldest Active Hint = 175

100	200	1		John	Smith	Kirkland
200	∞	1		John	Smith	Redmond
50	100	1		Jim	Spring	Kirkland
300	∞	1		Ken	Stone	Boston

Garbage Cleanup

- Non-blocking, Cooperative, Efficient, Responsive, Scalable
- Active transactions work cooperatively and pick up parts of GC work
- A dedicated system thread for GC
- GC thread wakes every minute or when the number of committed transactions since it last ran exceed 1024



GC Diagnostics

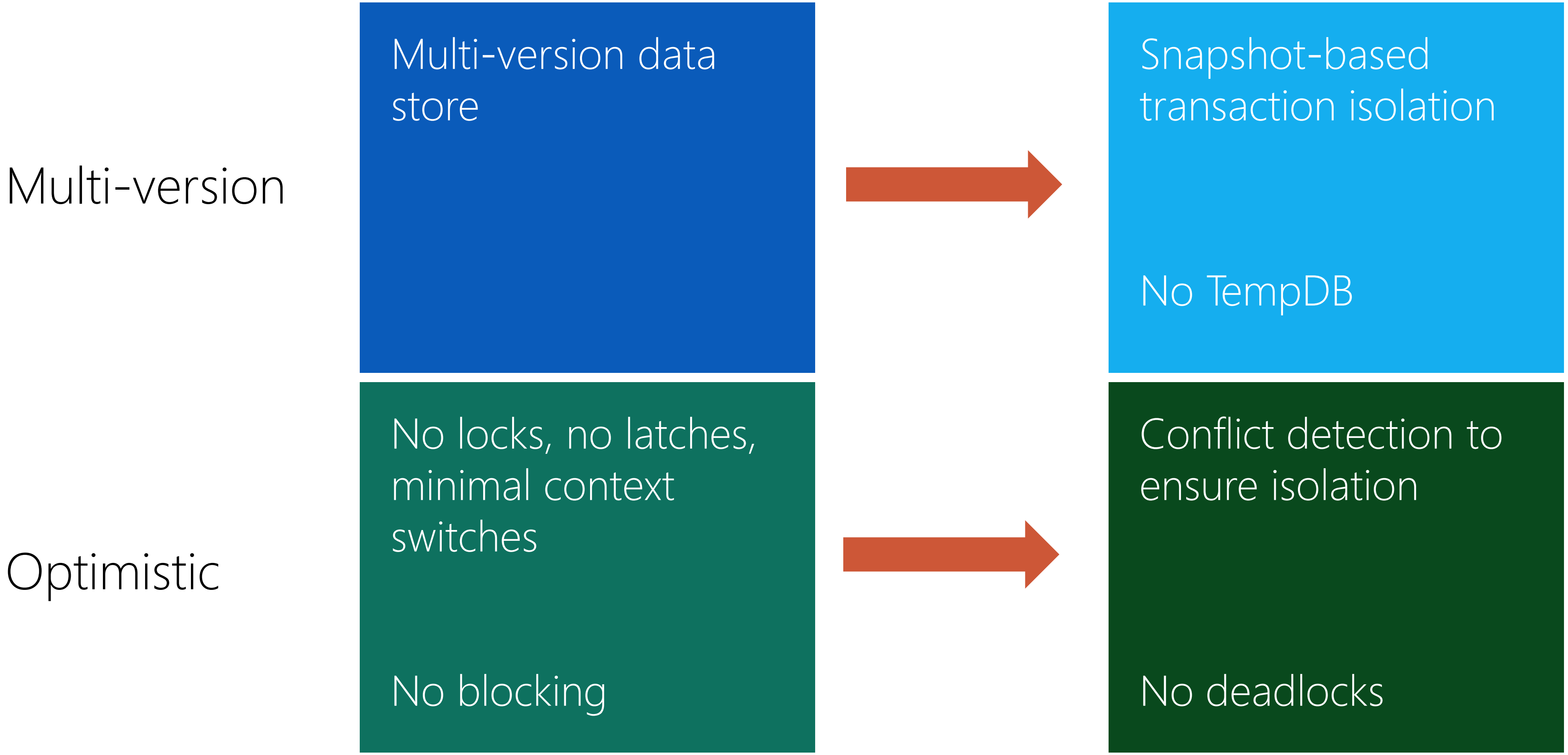
DMV’s

- sys.dm_xtp_gc_stats
- sys.dm_xtp_gc_queue_stats

Performance counters – XTP Garbage Collection

Extended Events – xtpengine.gc_*

Concurrency Control



Multi-version

Multi-version data store

Snapshot-based transaction isolation

No TempDB

Optimistic

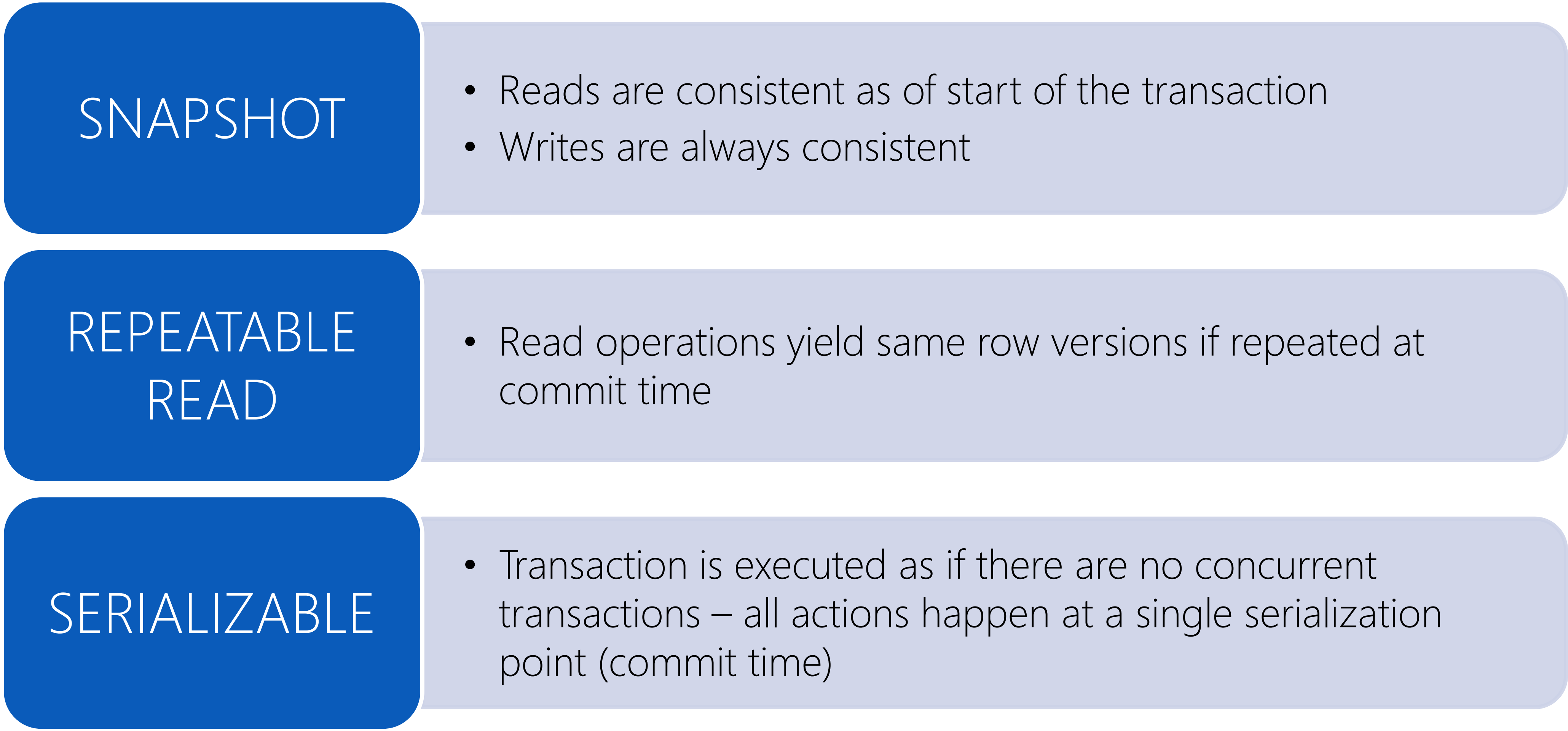
No locks, no latches, minimal context switches

No blocking

Conflict detection to ensure isolation

No deadlocks

Isolation Levels Supported



SNAPSHOT

- Reads are consistent as of start of the transaction
- Writes are always consistent

REPEATABLE READ

- Read operations yield same row versions if repeated at commit time

SERIALIZABLE

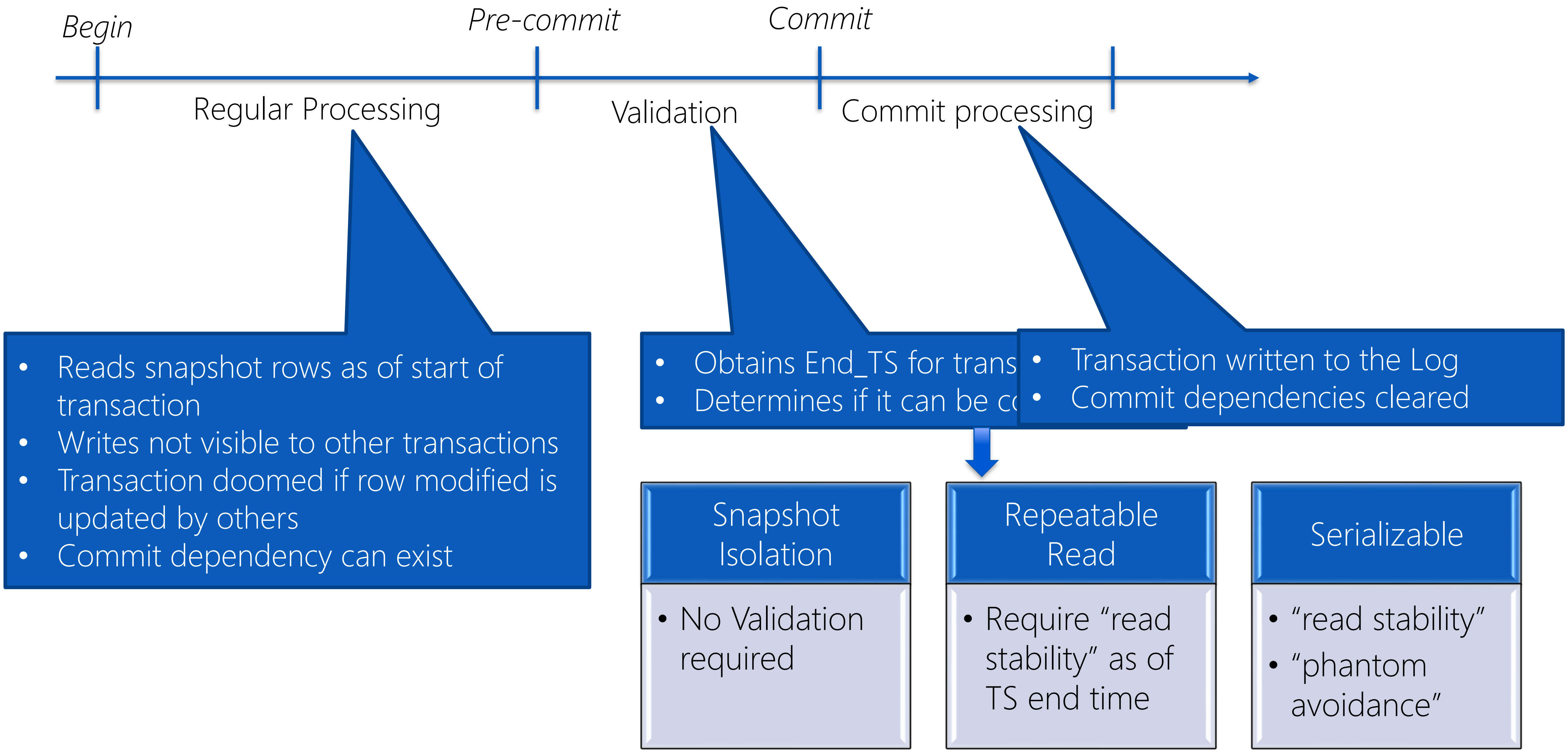
- Transaction is executed as if there are no concurrent transactions – all actions happen at a single serialization point (commit time)

Write Conflict

Time	Transaction T1 (SNAPSHOT)	Transaction T2 (SNAPSHOT)
1	`BEGIN`	
2		`BEGIN`
3		`UPDATE t SET c1='value2' WHERE c2=123`
4	`UPDATE t SET c1='value1' WHERE c2=123 (write conflict)`	

First writer wins

Transaction Processing



Begin

Pre-commit

Commit

Regular Processing

Validation

Commit processing

- Reads snapshot rows as of start of transaction
- Writes not visible to other transactions
- Transaction doomed if row modified is updated by others
- Commit dependency can exist

- Obtains End_TS for trans
- Determines if it can be co

- Transaction written to the Log
- Commit dependencies cleared

Snapshot Isolation

- No Validation required

Repeatable Read

- Require “read stability” as of TS end time

Serializable

- “read stability”
- “phantom avoidance”

Validation Errors and Retry Logic

Err Number	Err Message
41302	The current transaction attempted to update a record that has been updated since the transaction started
41305	The current transaction failed to commit due to a repeatable read validation failure
41325	The current transaction failed to commit due to a serializable validation failure
41301	A previous transaction that the current transaction took a dependency on has aborted, and the current transaction can no longer commit

Failures causing transaction abort

Write conflicts, validation failures

Aborted transactions need to be retried

Solution: implement retry logic

Server-side retry avoids changes to client apps

Retry Logic for Transaction Failures

```
CREATE PROCEDURE usp_my_procedure @param1 type1, @param2 type2, ...
AS
BEGIN
 DECLARE @retry INT = 10
 WHILE (@retry > 0)
 BEGIN
  BEGIN TRY

   EXEC usp_my_native_proc @param1, @param2, ...

   SET @retry = 0
  END TRY
  BEGIN CATCH
   SET @retry -= 1
   IF (@retry > 0 AND error_number() IN (41302, 41305, 41325, 41301, 1205))
    IF (@@TRANCOUNT>0) ROLLBACK TRANSACTION
   ELSE
    THROW
  END CATCH
 END
END
```

Hekaton-specific error codes

Deadlock (for disk-based tables)

Isolation Level Combinations Supported

✓

Disk-based	Memory optimized	Usage recommendations
READCOMMITTED	SNAPSHOT	• Baseline combination – most cases that use READCOMMITTED today
READCOMMITTED	REPEATABLEREAD/ SERIALIZABLE	• Data migration • In-Memory OTLP only Interop
REPEATABLEREAD/ SERIALIZABLE	SNAPSHOT	• Memory-optimized table access is INSERT-only • Useful for data migration and if no concurrent writes on memory-optimized tables (e.g., ETL)

Unsupported isolation level combinations (V1)

✗

Disk-based	Memory optimized
SNAPSHOT	Any isolation level
REPEATABLEREAD/ SERIALIZABLE	REPEATABLEREAD/ SERIALIZABLE