Optional Lessons

Module 10

Learning Units covered in this Module

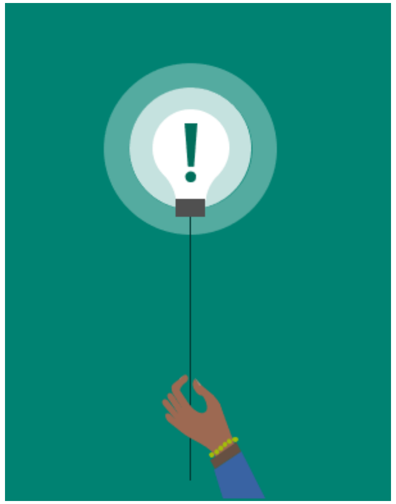
- Extra: Test Storage Subsystem with Diskspd
- Extra: Partitioned Tables and Indexes
- Extra: Columnstore Indexes
- Extra: In-Memory OLTP Tables
- Extra: Intelligent Query Processing
- Extra: Resource Governor

Extra: Test Storage Subsystem with Diskspd

Objectives

After completing this learning, you will be able to:

- Use tools to test the storage subsystem for SQL Server.



DiskSpd utility

Storage performance tool

Highly customizable I/O load generator tool

- Storage performance tests against:
  - files
  - partitions
  - physical disks.
- Generate a wide variety of disk request patterns for use in analyzing and diagnosing storage performance issues.
- Simulates SQL Server I/O activity, with detailed XML output for result analysis.

SQL Server disk usage patterns

R/W%	Type	Block	Threads / Queue	Simulates
80/20	Random	8K	# cores / Files	Typical OLTP data files
0/100	Sequential	60K	1/32	Transaction Log
100/0	Sequential	512K	1/16	Table Scans
0/100	Sequential	256K	1/16	Bulk load
100/0	Sequential	1MB	1/32	Backup
0/100	Random	64K-256K	# cores / Files	Checkpoints

Customizing tests for SQL Server

Parameter	Description
-c<size>	Create files of the specified size.
-d<seconds>	Duration
-f<rst>	Set random or sequential access hints.
-b<size>	The block size used for I/O operations.
-r<alignment>	turns on random I/O access.
-s[i]<size>	Sequential I/O parameter.
-w<percentage>	Perform a write test.
-F<count>	Total number of threads.
-t<count>	Number of threads per target.
-o<count>	Number of outstanding I/O requests per-target per-thread.
-Sh	Disable both software caching and hardware write caching.
-L	Measure latency statistics.

Sample command lines

- Review disk performance for sequential IO operations (TransactionLog)
  diskspd.exe -c50G -d20 –si –w100 –t8 –o8 –b60K -Sh -L F:\testfile.dat

- Review disk performance for random IO operations (OLTP)
  diskspd.exe -c50G -d20 -r –w80 –t8 –o8 -b64K -Sh -L F:\testfile.dat

- Review disk performance for sequential IO operations (Table Scan)
  diskspd.exe -c50G -d20 –si –w0 –t8 –o8 –b512K -Sh -L F:\testfile.dat

Analyzing test results

- DiskSpd provides per-thread per-target statistics on data read and written by each thread in terms of:

Total bytes

Bandwidth

IOPs

- Advanced sets of statistics which DiskSpd can optionally collect:

Latency

IOPs statistics

Demonstration

Executing DiskSpd and analyzing output

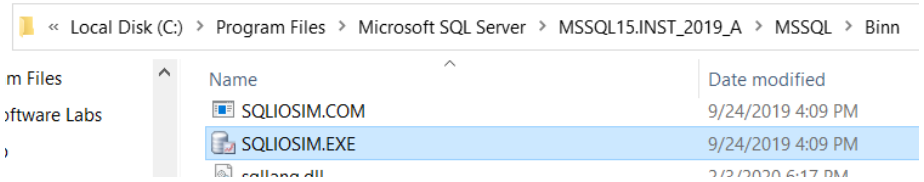
SQLIOSim

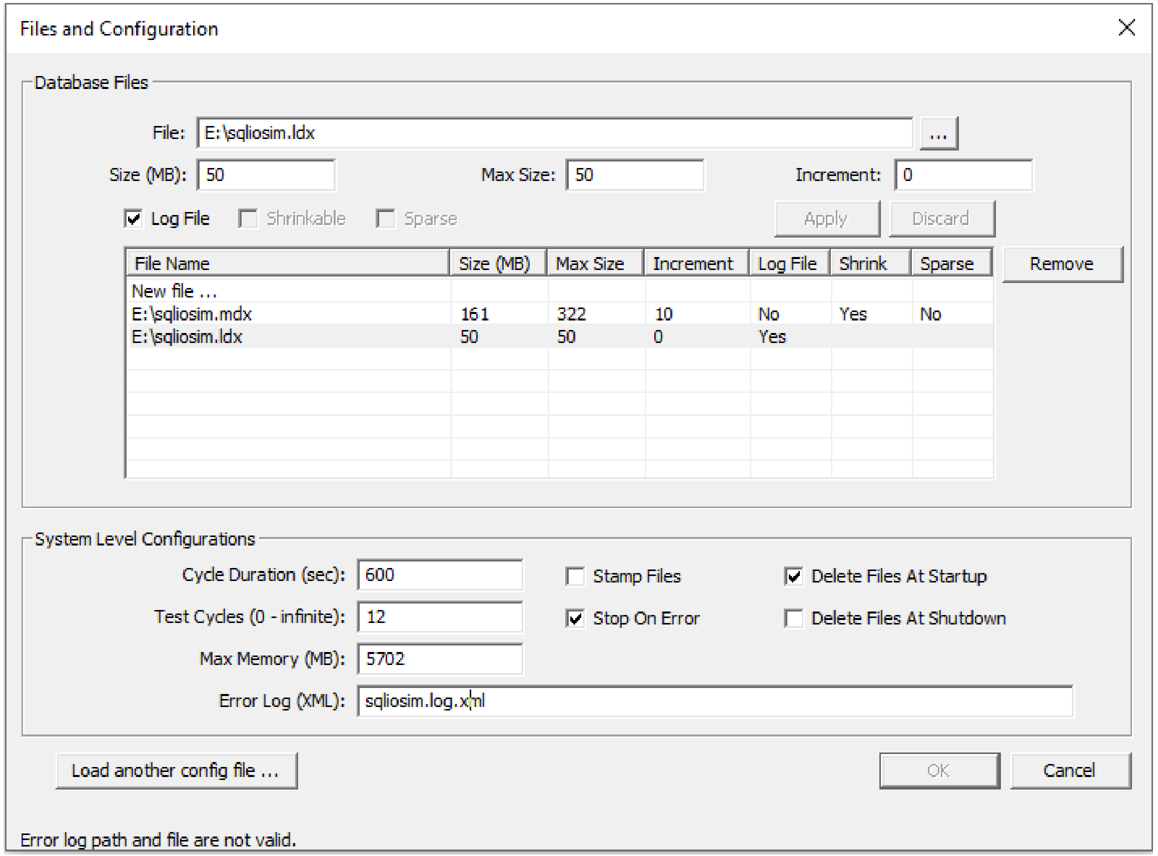
- Utility tool to perform reliability and integrity tests on disk subsystems.
- These tests simulate read, write, checkpoint, backup, sort, and read-ahead activities for Microsoft SQL Server.
- Used to test the I/O stability, not performance characteristics.

To perform benchmark tests and determine the I/O capacity of storage system use Diskspd tool.

SQLIOSim

- SQLIOSim.com and SQLIOSim.exe utilities are found in the BINN folder under
  - *C:\Program Files\Microsoft SQL Server\<instance_name>\MSSQL\BIN:*





- SQLIOSim.exe is a graphical application.

Demonstration

Executing SQLIOSim and analyzing output

DiskSpd

Using DiskSpd to obtain storage performance metric

LAB

Questions?

Knowledge Check

What tool should be used to perform benchmark tests and determine I/O capacity of the storage system?

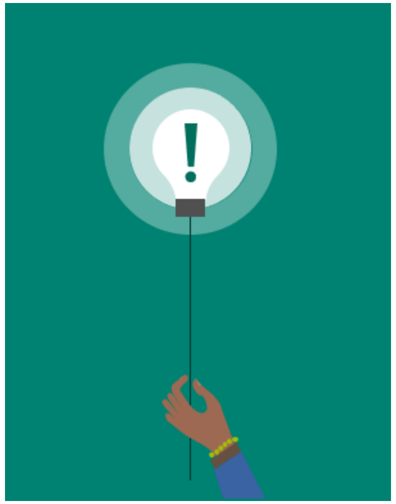
What tool should you use if you have to perform reliability and integrity tests on disk subsystems?

Extra: Partitioned Tables and Indexes

Objectives

After completing this learning, you will be able to:

- Understand Partition Concepts.
- Explain benefits of partitioning.
- Distinguish between aligned vs non-aligned indexes.
- Manage partitions.
- Apply Partition performance guidelines.



What is Partitioning?

"Horizontal Partition"

Table or Index Partition

- Breaking a single table or index into multiple HoBT (Heap or B-Tree) structures
- Requires a single column to use as partition key
- Partitioning is transparent to all queries

Partitioning benefits

Data Access

- Faster data load and offload.
- Query performance may be improved:
  - when predicate uses Partition-Key → partition elimination
  - when predicate doesn't use partition key → parallel execution

Maintenance and management

- Data placement handled by SQL Server.
- Piecemeal backup / restore of data by partition.
- Per-Partition management available for:
  - Index Maintenance
  - Lock Escalation
  - Compression

Horizontal Partitioning

Table without Partition

ProductKey	OrderDateKey	CustomerKey	SalesOrderNumber	UnitPrice	ExtendedAmount	ProductStandardCost
312	20050922	28191	SO44236	3578.27	3578.27	2171.2942
313	20050923	28163	SO44242	3578.27	3578.27	2171.2942
314	20050926	28142	SO44254	3578.27	3578.27	2171.2942
312	20050927	28171	SO44259	3578.27	3578.27	2171.2942

Partition Key

Range for Partition 1

Table with Partition

Partition 1

ProductKey	OrderDateKey	omerKey	SalesOrderNumber	UnitPrice	ExtendedAmount	ProductStandardCost
312	20050922	28191	SO44236	3578.27	3578.27	2171.2942
313	20050923	28163	SO44242	3578.27	3578.27	2171.2942

Range for Partition 2

Partition 2

ProductKey	OrderDateKey	Cu	rNumber	UnitPrice	ExtendedAmount	ProductStandardCost
314	20050926	28142	SO44254	3578.27	3578.27	2171.2942
312	20050927	28171	SO44259	3578.27	3578.27	2171.2942

Table Partitioning Concepts

Partition Function

- Defines how the table is logically partitioned.
- Defines data type of partition key, and ranges of each partition.
- Ranges can be modified as needed.

Partition Schemes

- Specify mapping from partitioned table to Filegroup(s).
- Uses *Partition Function* to perform mapping.
- Different filegroups can provide increased performance, but is not required.

Partition Column (key)

- Column used to apply partition function.
- If using a computed column, it must be explicitly marked as PERSISTED.

Partition Table & index

- Partitioned Tables are defined using *Partition Schemes*.
- Several tables may share common Partition Functions and Partition Schemes.
- Loading data into the table or index creation, will move data to correct partition automatically.

Aligned vs Non-aligned indexes

Aligned Index

- An index that is built on the same partition scheme as its corresponding table.
- When a table and its indexes Align, SQL Server can switch partitions quickly and efficiently.

Non-aligned index

- An index partitioned independently from its corresponding table.
- Designing a non-aligned partitioned index can be useful in the following cases:
  - The base table has not been partitioned.
  - The index key is unique, and it does not contain the partitioning column of the table.
  - Useful when want to joins with more tables using different join columns.

Indexes on Partitioned Table

Logical Alignment

- Indexes are logically aligned when they use partitioning column.

Storage Alignment

- Indexes are storage aligned when created on the same partitioning scheme.
- By default, indexes will be created ON the partition scheme – 'storage aligned' indexes.

```
CREATE [CLUSTERED] INDEX … ON {partition_scheme_name ( column_name ) }
```

Partitioned table and index sample

```sql
-- Creating a partition function with four partitions
CREATE PARTITION FUNCTION myRangePF1 (int)
    AS RANGE LEFT FOR VALUES (1, 100, 1000) ;
GO
-- Creates a partition scheme that applies myRangePF1 to the four filegroups
CREATE PARTITION SCHEME myRangePS1
    AS PARTITION myRangePF1
    TO (test1fg, test2fg, test3fg, test4fg) ;
GO
-- Creates a partitioned table that uses myRangePS1 to partition col1
CREATE TABLE PartitionTable (col1 int NOT NULL, col2 char(10))
    ON myRangePS1 (col1) ;
GO
CREATE CLUSTERED INDEX ClusIndxCol1 ON PartitionTable(col1)
ON myRangePS1(col1)
GO
```

Table Partitioning Operations

Data Movement in table or Partition is done using ALTER ...SWITCH/MERGE/SPLIT

SWITCH	MERGE	SPLIT
• Uses staging table • MUST have same schema as Partitioned table	• Used to eliminate an empty partition, by merging it with another partition	• Used to create a new empty partition from an existing partition

```
--SWITCH
ALTER TABLE PartitionTable SWITCH PARTITION 2 TO NonPartitionTable ;
--SPLIT
CREATE PARTITION FUNCTION myRangePF1 (int) AS RANGE LEFT FOR VALUES ( 1, 100, 1000 );
ALTER PARTITION FUNCTION myRangePF1() SPLIT RANGE (500);
--MERGE
CREATE PARTITION FUNCTION myRangePF1 (int) AS RANGE LEFT FOR VALUES ( 1, 100, 1000 );
ALTER PARTITION FUNCTION myRangePF1() MERGE RANGE (100);
```

Demonstration

Partitioning

Explore Partition Schemes and Functions.

SQL Server Partitioning

- Practicing with a sliding partition window

Questions?

Knowledge Check

What three structures are required to support partitions?

What are some benefits of partitioning?

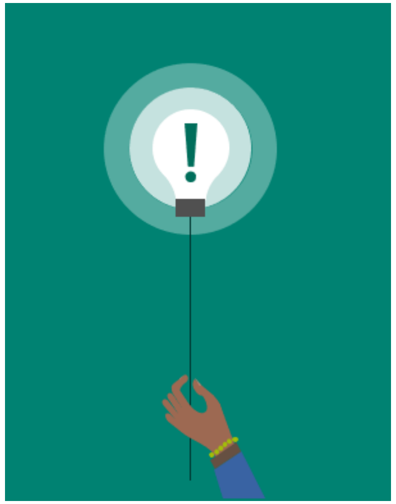
What is used for data movement in partitions or table?

Extra: Columnstore Indexes

Objectives

After completing this learning, you will be able to:

- Understand the differences between rowstore and Columnstore indexes.
- Understand the primary use case for Columnstore indexes.
- Examine and monitor Columnstore indexes.

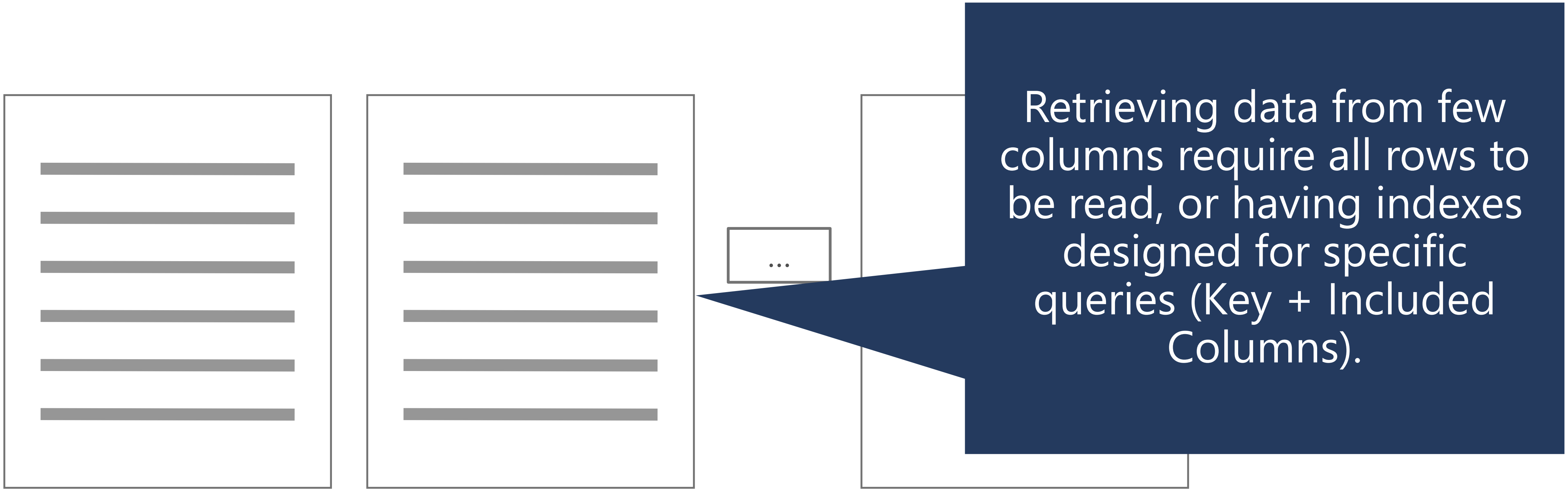


Rowstore INDEX design

- Up to 1,000 indexes per table
- Up to 16 columns for index key
- 900 bytes per clustered index
- 1700 bytes per non-clustered index

Regular types of indexes (Cluster/NonClustered) wouldn’t be able to handle the workload.
It will be impractical to create indexes on all individual columns, or to create all possible column combinations.

Traditional Storage Models



Why Columnstore indexes?

Clustered Index	Non-Clustered Index
• Use to store fact tables and large dimension tables for data warehousing workloads. • This method improves query performance and data compression by up to 10 times.	• Use to perform analysis in real time on an OLTP workload.

Why should I use a Columnstore index?

Columns store values from the same domain and commonly have similar values, which result in high compression rates. I/O bottlenecks in your system are minimized or eliminated, and memory footprint is reduced significantly.

High compression rates improve query performance by using a smaller in-memory footprint. In turn, query performance can improve because SQL Server can perform more query and data operations in memory.

Batch execution improves query performance, typically by two to four times, by processing multiple rows together.

Queries often select only a few columns from a table, which reduces total I/O from the physical media.

Columnstore Index Storage Model

- Each page stores data from a single column
- Highly compressed
  More data fits in memory
- Each column can be accessed independently
  Fetch only columns that are needed
  Can dramatically decrease I/O

Row Groups & Segments

Segment

Contains values for one column for a set of rows.

Segments are compressed.

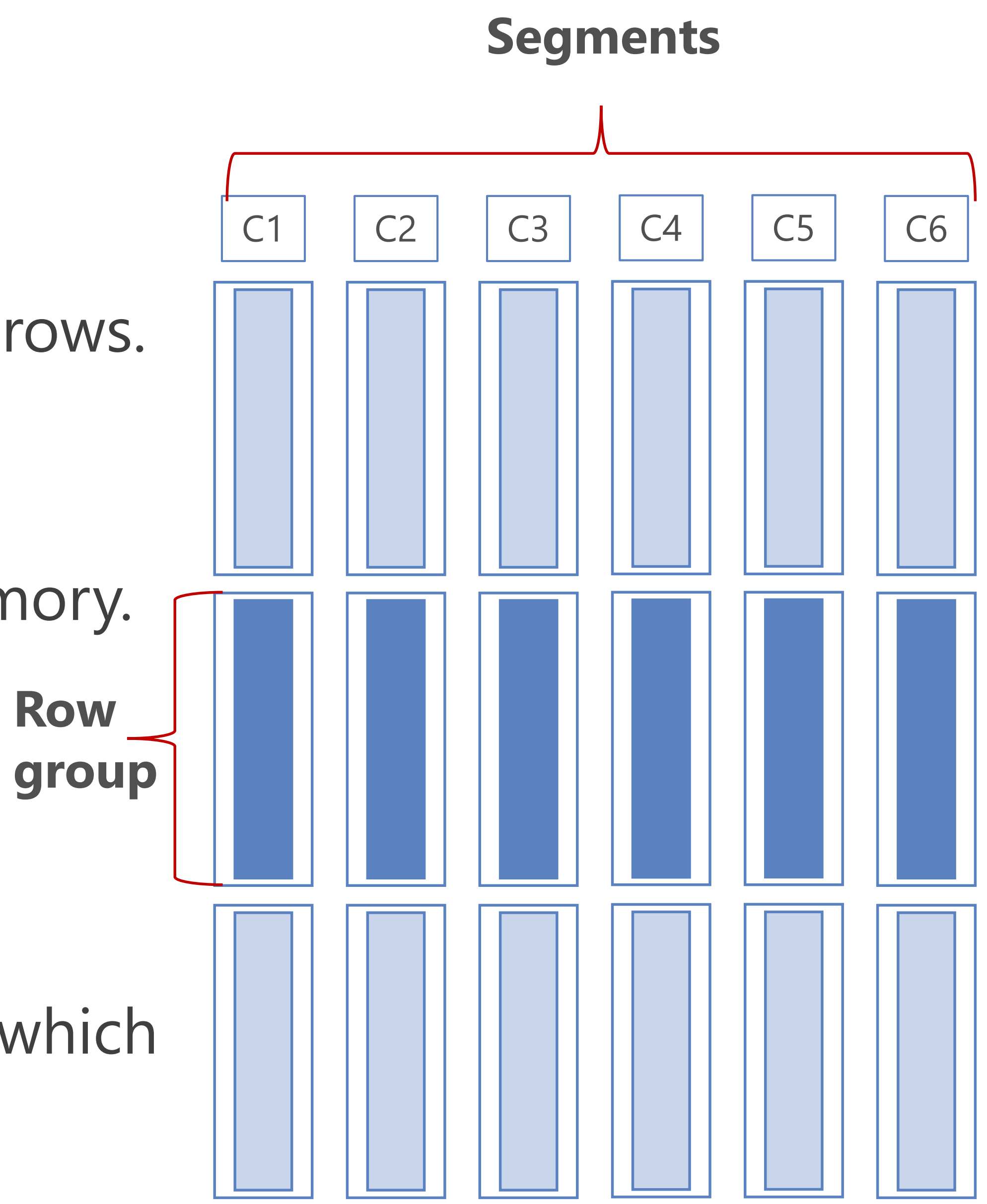
Each segment is stored in a separate LOB.

It is a unit of transfer between disk and memory.

Row Group

Segments for the same set of rows comprise a row group.

Position of a value in a column indicates to which row it belongs to.



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
2010110	103	04	2	1	17.00
201011	109	04	2	2	20.00
201	103				17.0
2	106				

					mount
					25.00
20101108	109				10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

RowGroups are compress
column-wise manne

Number of rows per RowGroup is calculated as high as possible to get good compression rate, but low enough to not use too much memory

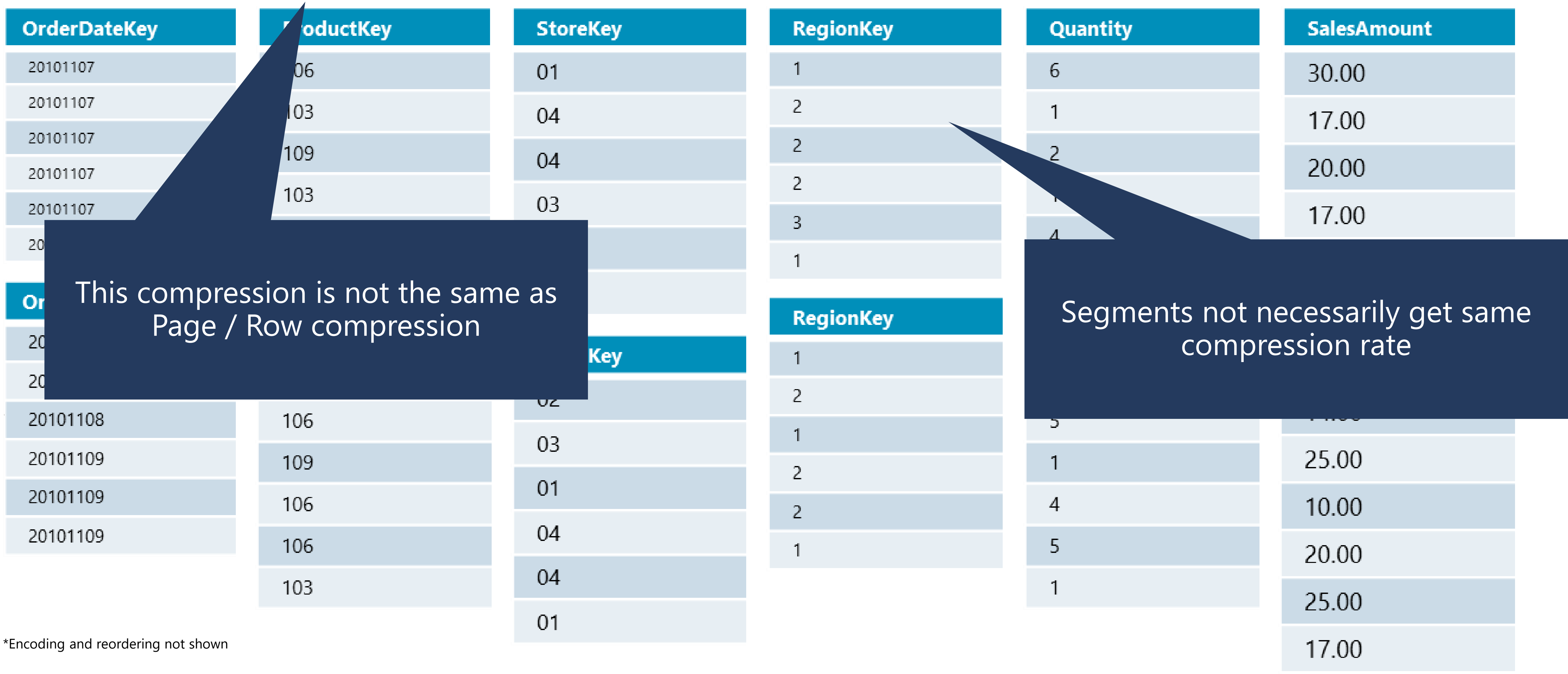
roup of Rows,
s

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



*Encoding and reordering not shown

Delta Store

A b-tree staging area

Rows accumulate until there are enough to be compressed into the columnstore format

- 1,048,576 rows
- A background process compresses these rows to columnstore compression

Allows for efficient access to “warm” data

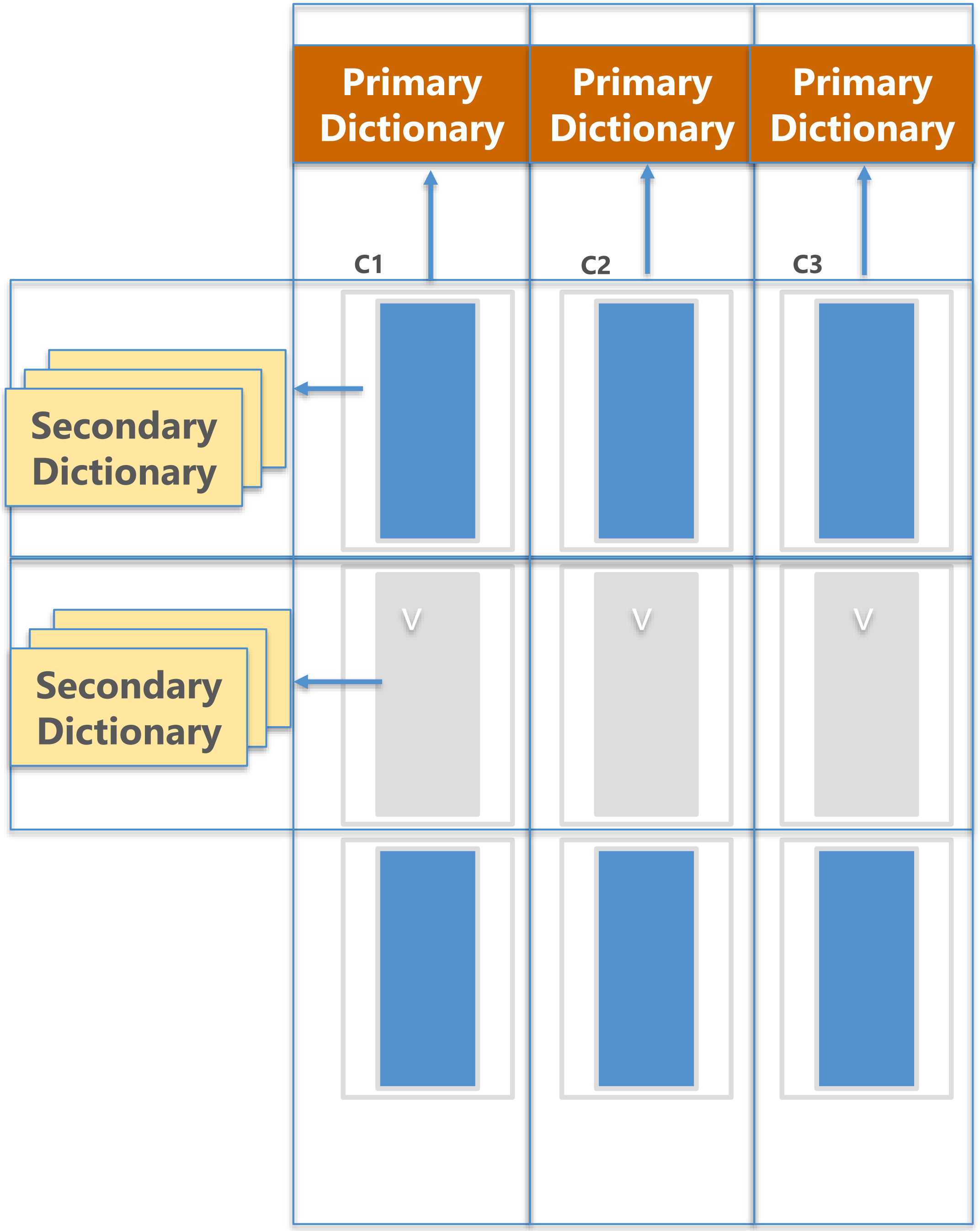
Improves compression of the columnstore index

Will not see batch mode execution from row store storage

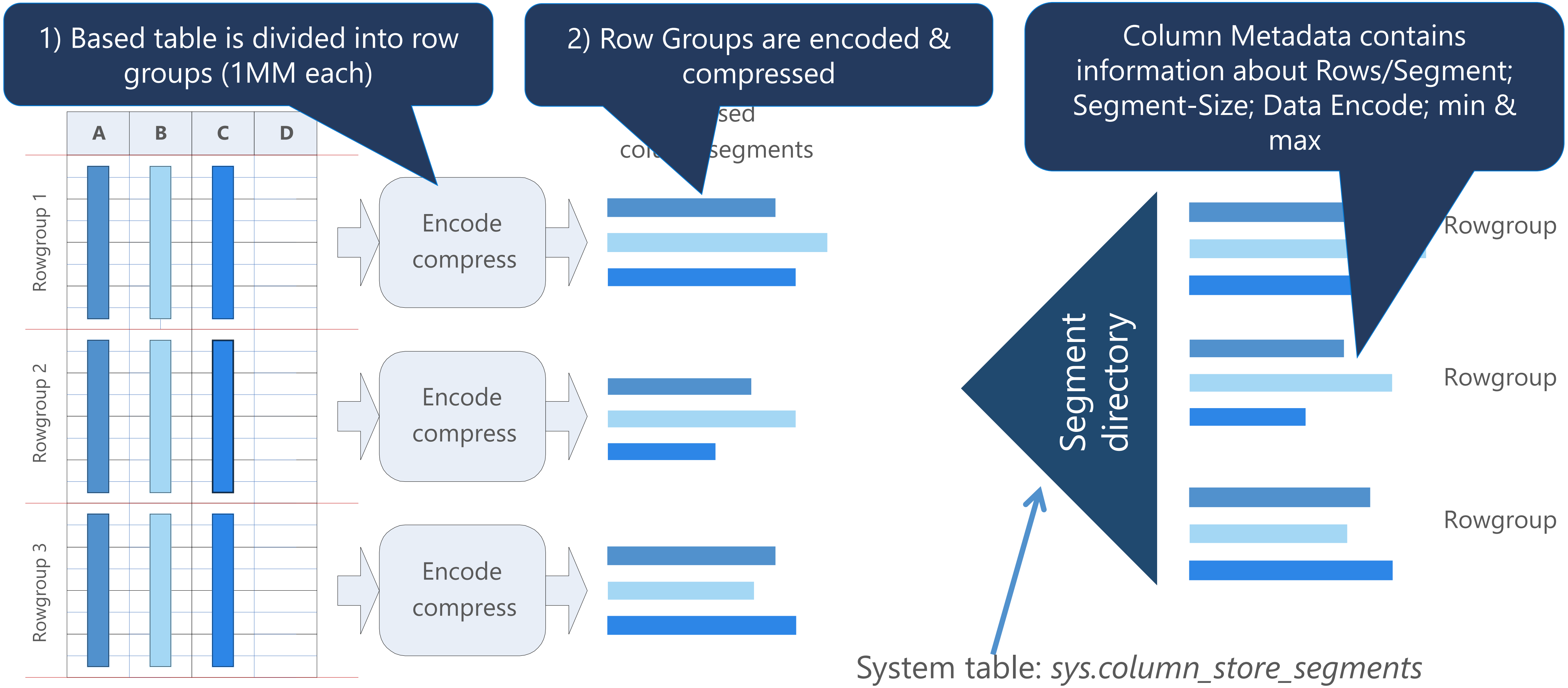
Introduced for clustered columnstore indexes but also exists for updateable non-clustered Columnstore indexes in 2016

Dictionaries

- String data encoded per column
- Encoded values in segments are bookmarks to dictionary
- Primary dictionary is one per partition
- Secondary dictionary is 1 or more per segment
- Stored as a LOB value as part of the table storage



Index Creation and Storage



System table: *sys.column_store_segments*
Includes segment metadata: size, min, max, …

Memory Requirements

Build/reindex is always parallel unless memory is constrained.

- **Can be FORCED with (MAXDOP = 1)**.

Roughly it takes about 1.5 times as long to create a columnstore index as compared to a b-tree index.

Memory required depends on the number of columns, types of columns, and amount of parallelism involved.

MAXDOP may be adjusted for the build as dictated by the amount of available memory.

Always occurs in batch mode in SQL 2016.

Batch Mode Processing

Process around 1000 rows at a time

As opposed to 1 row at a time with row-based processing

Not all query plan operators can perform batch processing

- nested loops
- merge join

Plan operators are expanded in SQL Server 2016+

Greatly reduced CPU time (7 to 40X)

Segment Elimination

Skips large chunks of data to speed up scans

Each partition in a column store index is broken into segments

Each segment has metadata that stores the minimum and maximum value of each column for the segment

The storage engine checks filter conditions against the metadata

If it detects no rows that qualify, it skips the entire segment without reading it from disk

Segment Elimination

```sql
SELECT ProductKey, SUM(SalesAmount)
FROM SalesTable
WHERE OrderDateKey < 20101108;
```

1. Fetches only needed columns

Not included in the query column list

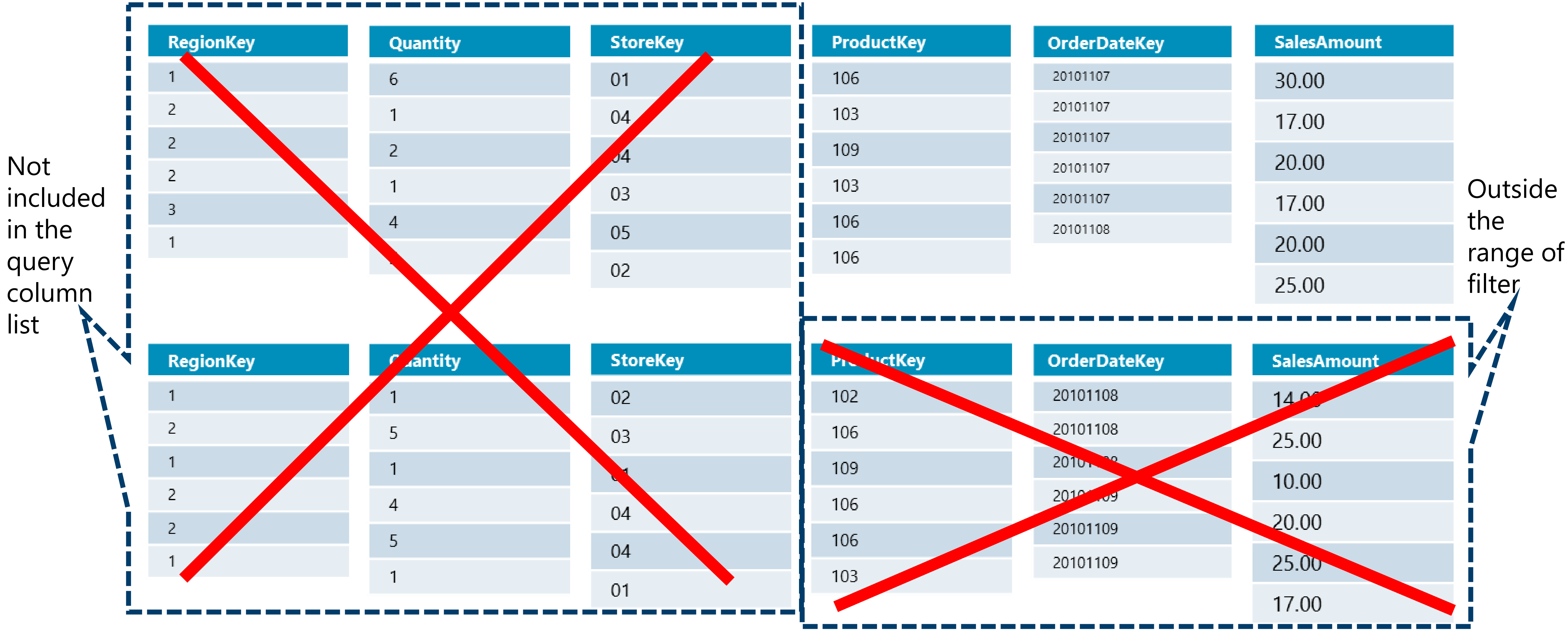
RegionKey	Quantity	StoreKey	ProductKey	OrderDateKey	SalesAmount
1	6	01	106	20101107	30.00
2	1	04	103	20101107	17.00
2	2	04	109	20101107	20.00
2	1	03	103	20101107	17.00
3	4	05	106	20101107	20.00
1	[illegible]	02	106	20101108	25.00

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

Fetch Only Needed Segments

```sql
SELECT ProductKey, SUM(SalesAmount)
FROM SalesTable
WHERE OrderDateKey < 20101108;
```

2. Fetches only needed segments



Monitoring Column Store Indexes

sys.column_store_row_groups

sys.column_store_segments

sys.column_store_dictionaries

Demonstration

Column store metadata

ColumnStore Index Performance

- Measure the performance of rowstore indexes versus colunmstore indexes.

Questions?

Knowledge Check

Name a primary reason for implementing a column store index?

What is the purpose of segments?

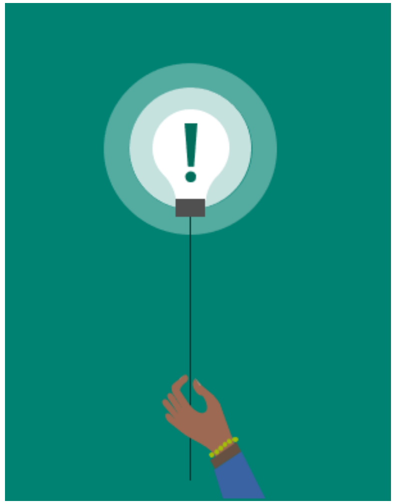
What is the purpose of row groups?

Extra: In-Memory OLTP

Objectives

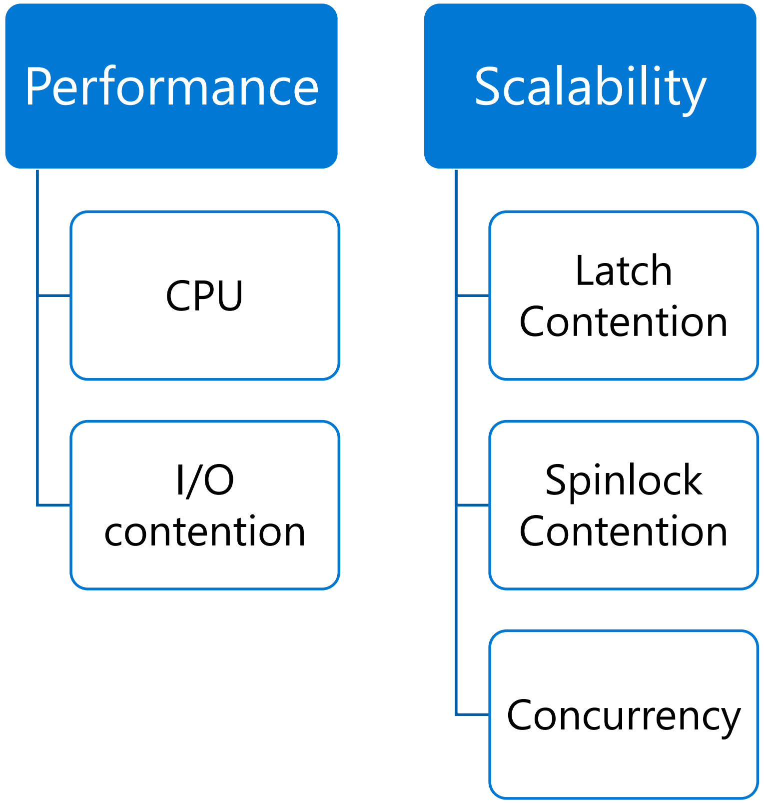
After completing this learning, you will be able to:

- Understand the motivation behind Memory-Optimized objects.
- Describe the characteristics of Memory-Optimized objects.
- Monitor resources consumed by Memory-Optimized objects.



Drivers for Memory-Optimized objects

With increased hardware capacity (CPU/memory/storage) new method of concurrent data access is required to address:



Memory-Optimized objects

Recommended use

Use to get:	Do not use when:

• Increased transaction throughput
• Improved data ingestion rate
• Reduced latency
• Transient data scenarios
• No locking
• No latching
	• Analytics/BI
	(consider Columnstore instead)
	• Bottleneck is outside SQL:
	  • Chatty App
	  • Bottleneck is in the app
	• Low transaction volume
	• Resource limitations – not enough memory

Memory-Optimized Objects

Object types

Objects

- Tables
- Types

Code

- Natively compiled Stored Procedures
- Triggers
- Functions

Memory-Optimized Objects

Can be accessed via:

- Interop T-SQL allows for full range of SQL syntax
- Natively compiled Stored Procedures

Transactions involving Memory-Optimized tables are Atomic, Consistent, Isolated, and Durable (ACID)

- Primary store for memory-optimized tables is main memory.
- All operations on these objects happen in memory.
- Support durable tables with Transaction Durability Delayed.

Durable

- Second copy of the table is maintained on disk

Non-Durable

- Not logged
- Data is not persisted on disk.

Memory-Optimized structure

Rows

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

Indexes

- Every memory-optimized object is required to have at least 1 index
- Indexes do not exist on disk.
  Only in memory.
  Recreated during recovery / restart
- Index types:
  - Hash indexes for point lookups
  - Range indexes (non-clustered) for ordered scans and Range Scans
  - There is no clustered Index
  - Indexes point to rows, access to rows is via an index

Memory-Optimized

Create Table DDL

```
CREATE TABLE [Customer](
    [CustomerID] INT NOT NULL
        PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT = 1000000),
    [Name] NVARCHAR(250) NOT NULL,
    [CustomerSince] DATETIME NULL
        INDEX [ICustomerSince] NONCLUSTERED
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Hash Index

BUCKET_COUNT 1-2X nr of unique index key values actual count is the next integer power of 2

Indexes are specified inline

This table is memory optimized

This table is durable

Non-durable tables:

DURABILITY=SCHEMA_ONLY

Memory-Optimized

Create Stored Procedure DDL

```
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

Memory-Optimized

Create Function DDL

```
CREATE FUNCTION [dbo].[ufnGetAccountingEndDate_native]()
RETURNS [datetime]
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT,
LANGUAGE=N'us_english')

    RETURN DATEADD(millisecond, -2, CONVERT(datetime, '20160701', 112));

END
```

Memory-Optimized Row Format

Row versioning

the actual "ROW"

Row Header	Row

Begin Ts	End Ts	StmtID	IdsLinkCount	Index1 ptr	Index2 ptr
8 bytes	8 bytes	4 bytes	2 bytes + 2 for padding	8 bytes * Number of Indexes	

Begin/End timestamps determine row's version validity and visibility

TimeStamps are 64bit BIGINT

BEGIN Ts:
Uses GLOBAL Timestamp if inserted within a transaction, otherwise it uses TRANSACTION-ID

END Ts:
Initially when row is inserted it contains a special value 'INFINITY'. When row is DELETED it uses GLOBAL Timestamp if deleted within a transaction, otherwise it uses TRANSACTION-ID

Statement ID:
Statement ID that created ROW.

Internal counters used to manage timestamp values:

- **TRANSACTION-ID**: reset when instance is restarted. Incremented with every new transaction
- **GLOBAL**: not reset on SQL restart; Incremented each time a transaction ends
- **The oldest active running Transaction** in the system

Memory-Optimized

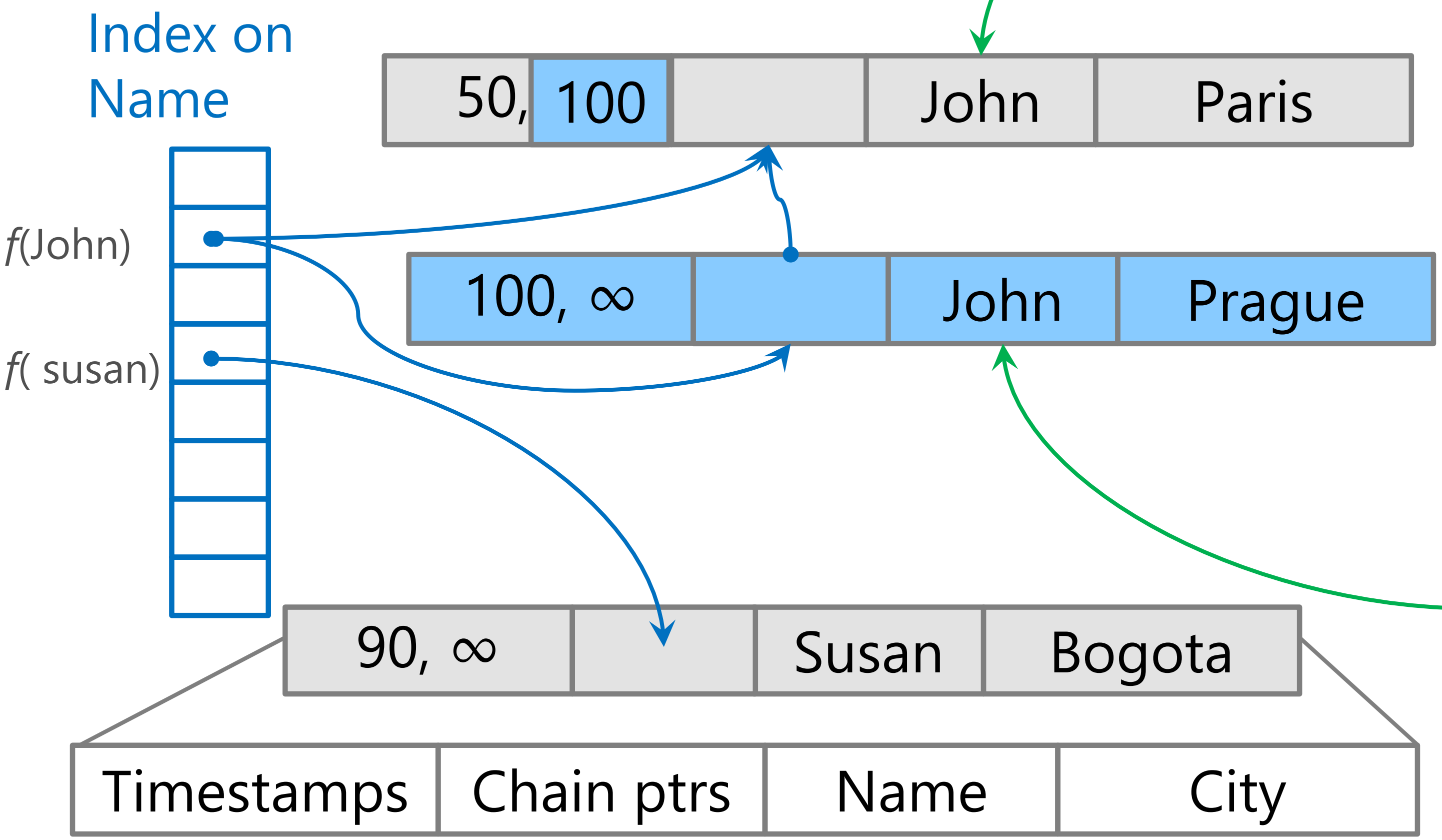
Row versions

- Rows are versioned
- Allows for concurrent reads and writes on the same row

Transaction 99: executes a SELECT

SELECT City
FROM <table>
WHERE Name = 'John'

Index seek returns pointer to row



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

Transaction 100:

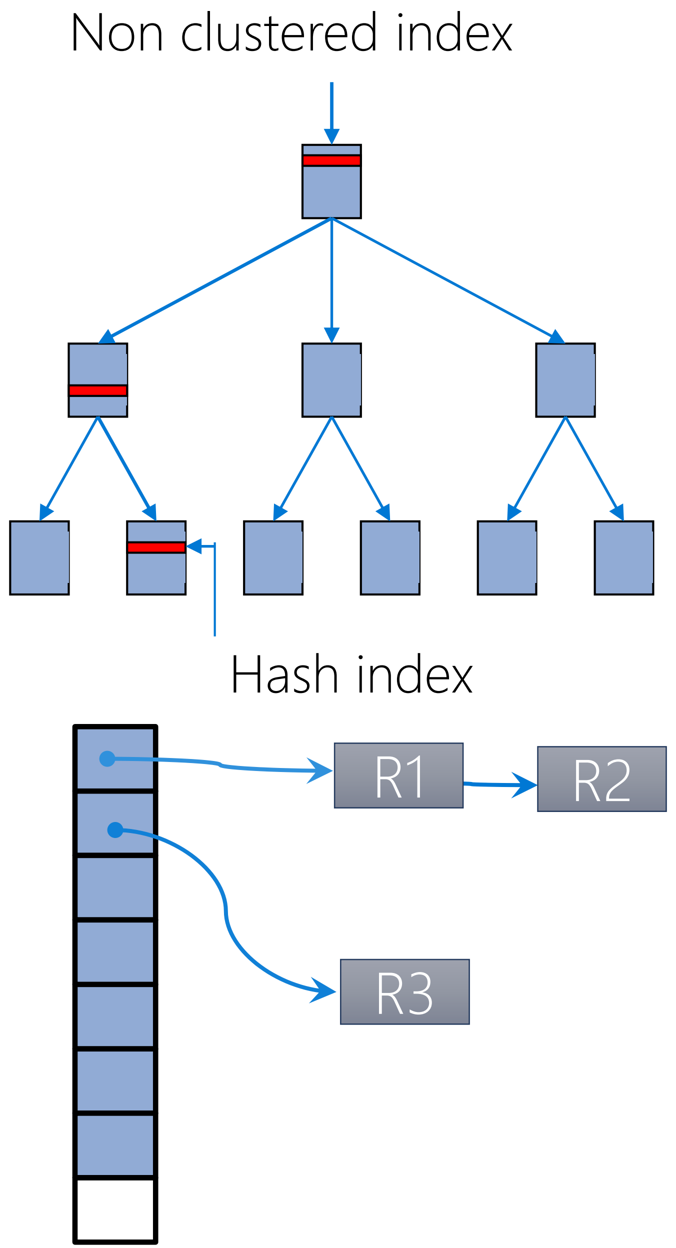
UPDATE <table>
SET City = 'Prague'
WHERE Name = 'John'

no locks of any kind

no interference with transaction 99

Memory-Optimized Indexes

- Requirement: at least 1 index per table
- Memory structure that connects the rows
- Exists only in memory
  are not persisted to disk
- Loaded during table creation, or at server startup
- Inherently covering
- Fragmentation and fill-factor not applicable
- No logging for index operations



Memory-Optimized

Index Types

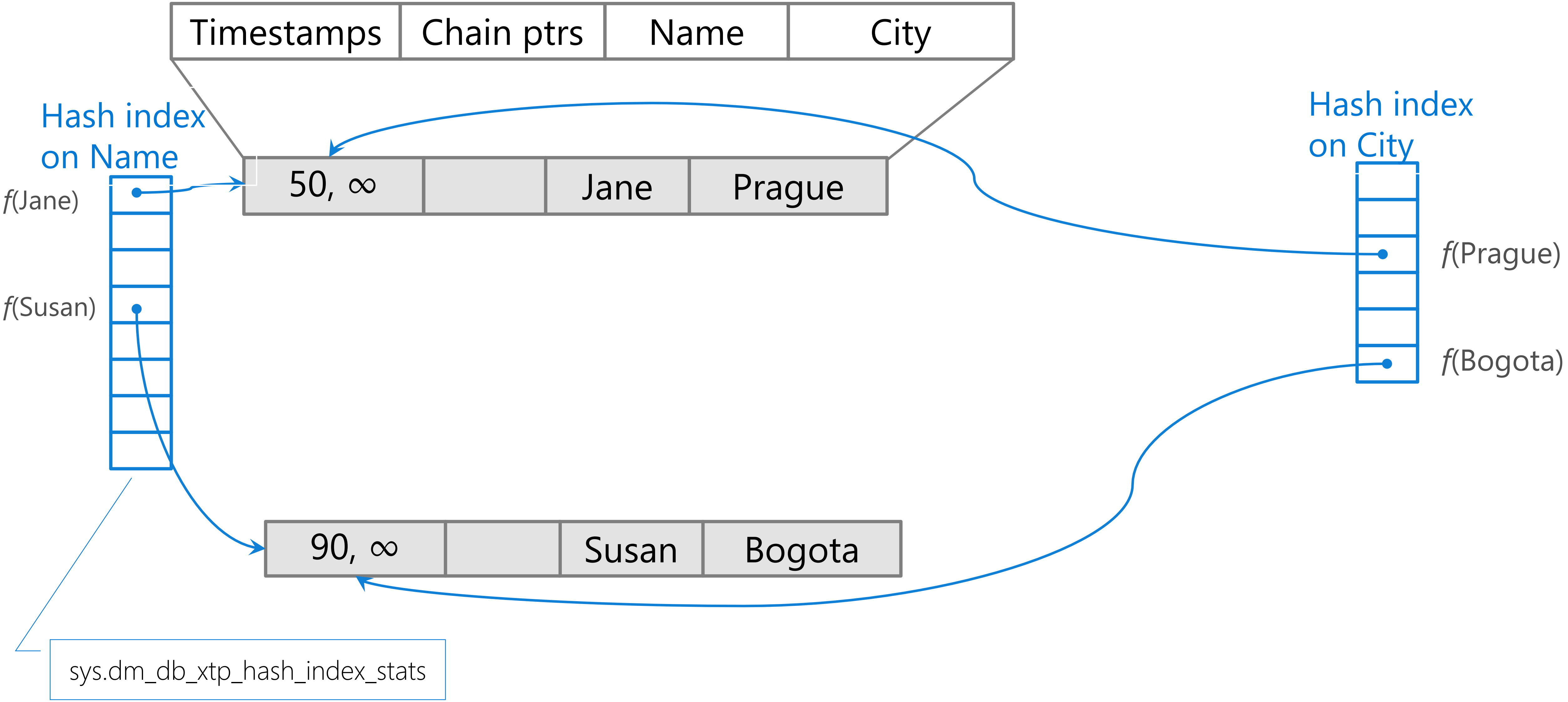
NonClustered	Hash
• Identical behavior as NonClustered index for disk-based tables • Resultset may be sorted by the NonClustered index	• Uses a function to map values to buckets • Each bucket entry (8-byte) is a pointer to row(s) • Collisions happen when hash-function returns same HASH for different rows • Resultset is never sorted

Memory-Optimized

Hash Index

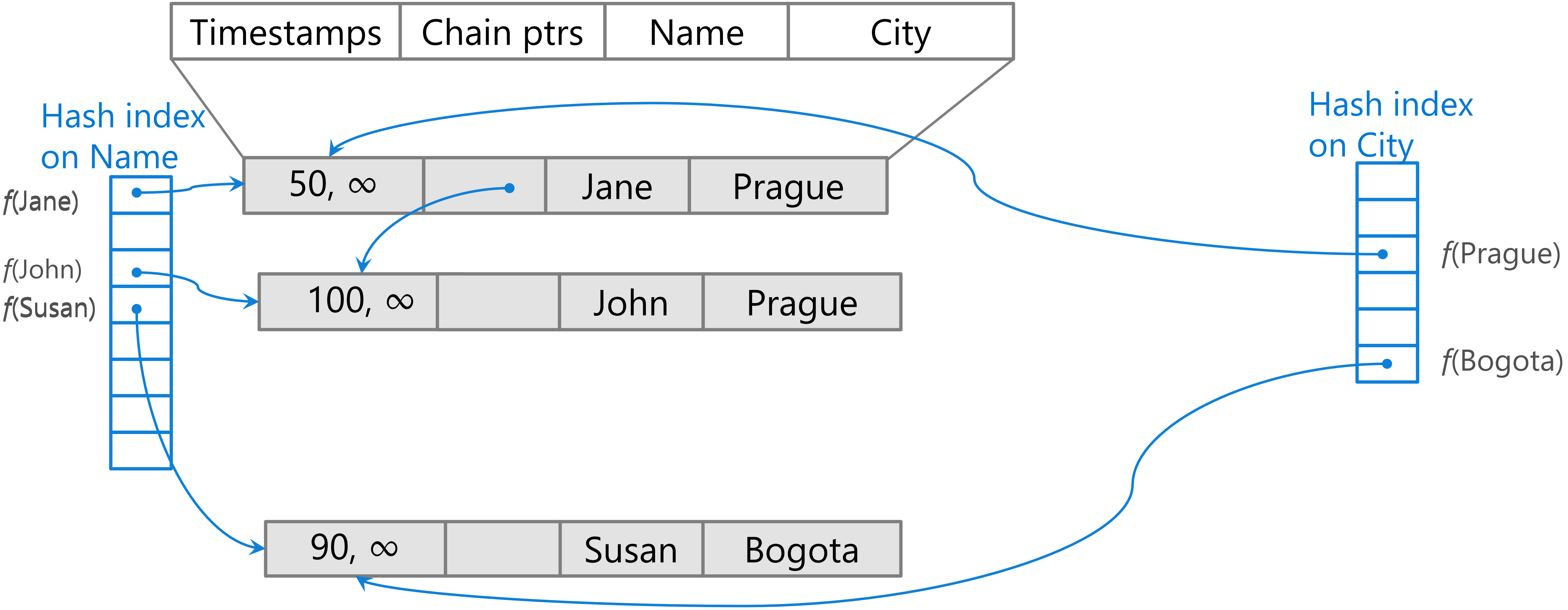
Bucket Count			
Specifies the number of unique entries • 8 bytes per entry	Small bucket count • conserves memory • May cause collisions	Large bucket count • High memory consumption • May avoid collisions	Best practices • Start with bucket count between 1 to 2 times the number of rows. • Monitor for duplicate entries. Adjust as necessary.

Traversing the Hash Index



Inserting a new row

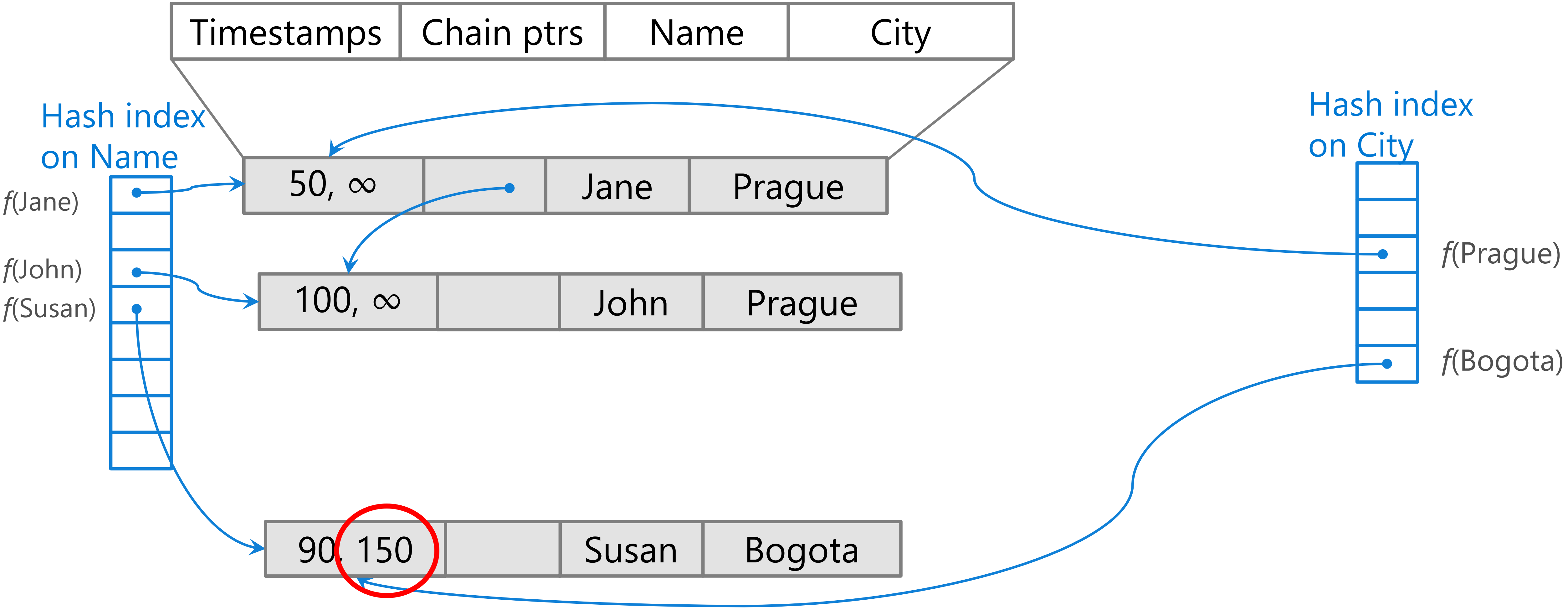
Insert



T100: INSERT (John, Prague)

Deleting a row

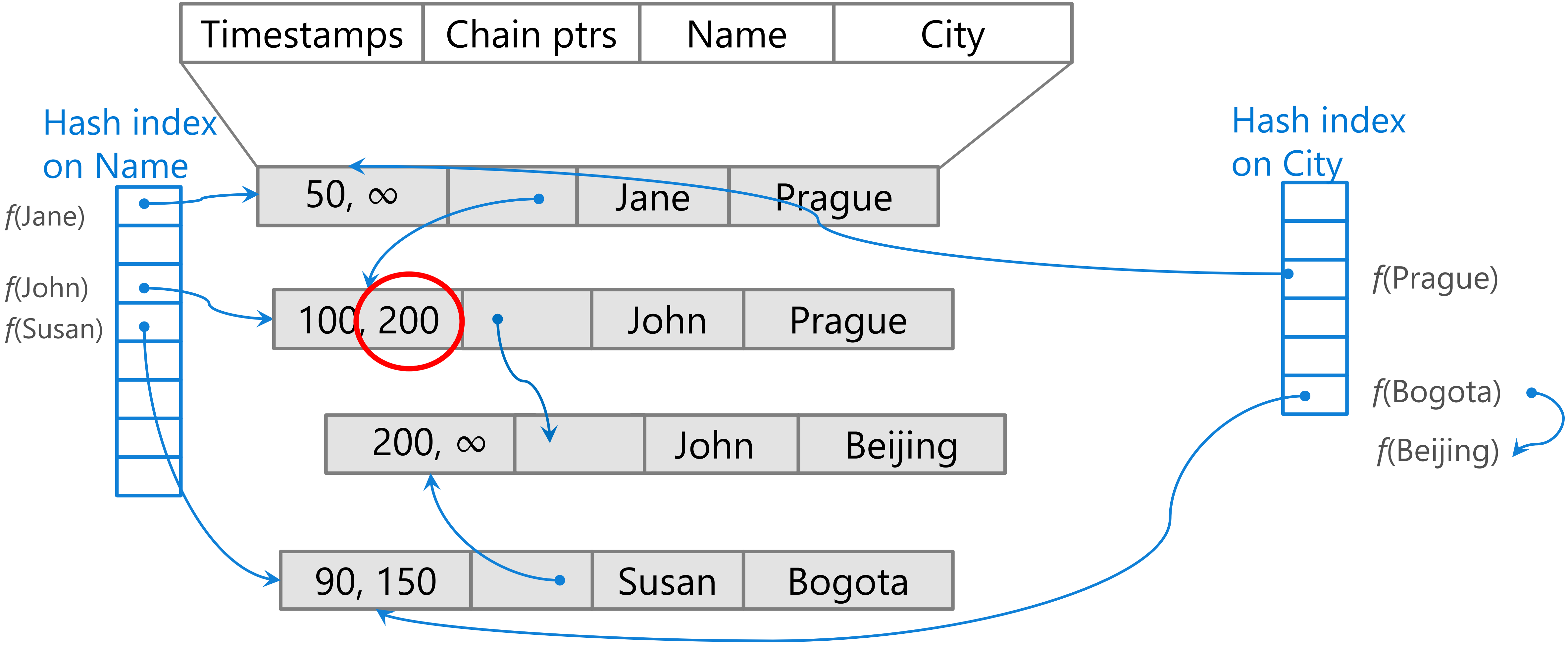
Delete



T150: DELETE (Susan, Bogota)

Updating a row

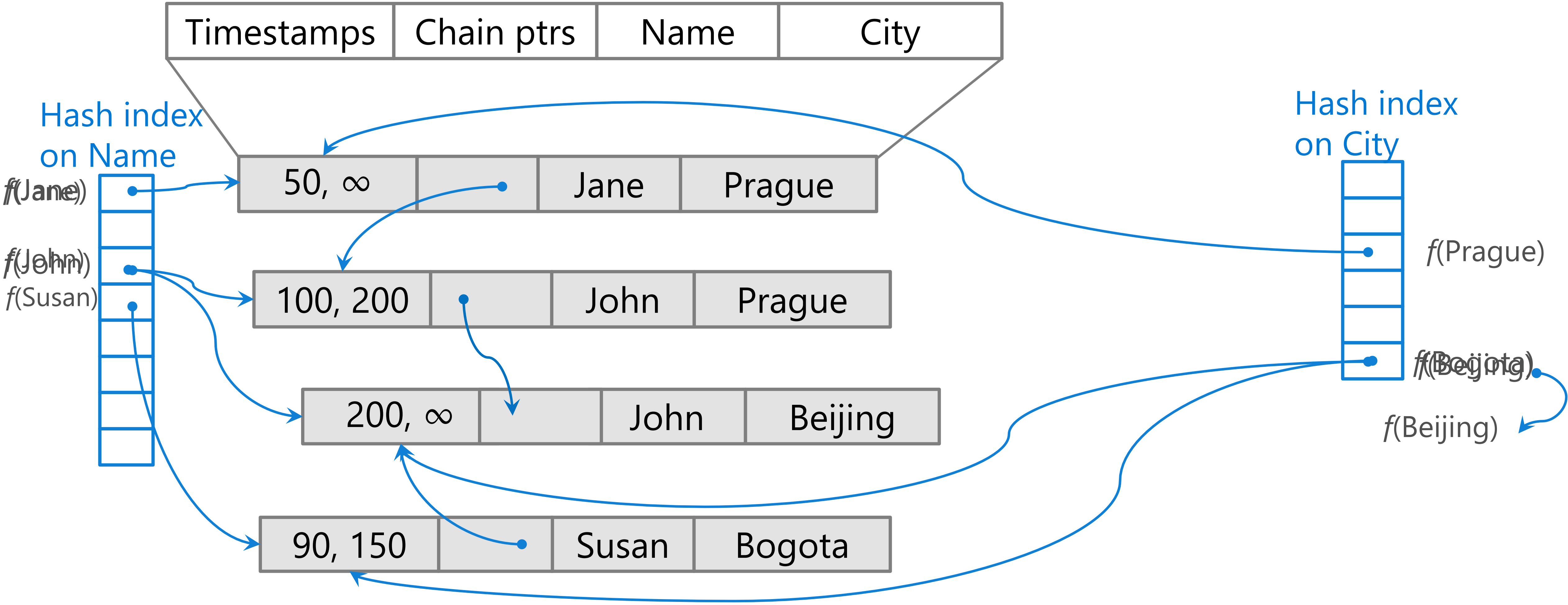
Update



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

Garbage Collection

Memory Clean-Up



T250: ***Garbage collection***

Demonstration

Memory-Optimized Objects

- Adding filegroup to support Memory-Optimized objects
- Creating Memory-Optimized objects
- Access Methods for Hash and NonClustered Indexes

Memory-Optimized

Troubleshooting Hash Index bucket count

- If the hash indexed values have a high rate of duplicates, the hash buckets suffer longer chains.

```
-- Calculate ratio of: Rows / Unique_Values.
DECLARE @allValues float(8) = 0.0, @uniqueVals float(8) = 0.0;

SELECT @allValues = Count(*) FROM <table>;
SELECT @uniqueVals = Count(*) FROM
          (SELECT DISTINCT <hash-key column> FROM <table>) as d;
-- If (All / Unique) >= 10.0, use a nonclustered index, not a hash.
SELECT Cast((@allValues / @uniqueVals) as  float) as [All_divby_Unique];
```

Memory-Optimized

Monitoring chain-length and empty buckets

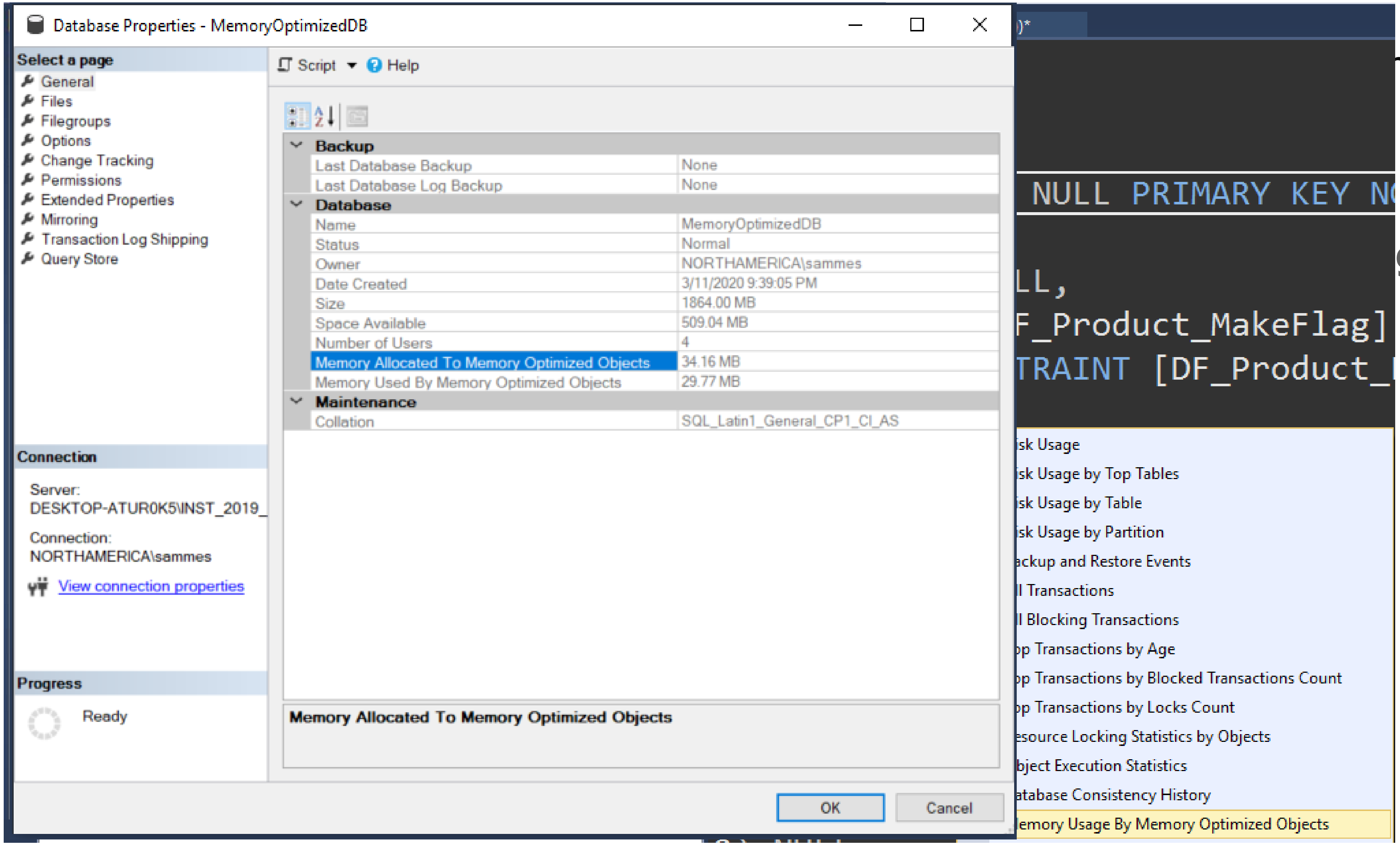
```
SELECT
  QUOTENAME(SCHEMA_NAME(t.schema_id)) + N'.' + QUOTENAME(OBJECT_NAME(h.object_id)) as [table],
  i.name as [index], h.total_bucket_count, h.empty_bucket_count,
  FLOOR((CAST(h.empty_bucket_count as float)/h.total_bucket_count) * 100)  as [empty_bucket_percent],
  h.avg_chain_length, h.max_chain_length
FROM
        sys.dm_db_xtp_hash_index_stats  as h
        JOIN sys.indexes as i
          ON h.object_id = i.object_id
          AND h.index_id  = i.index_id
        JOIN sys.memory_optimized_tables_internal_attributes ia ON h.xtp_object_id=ia.xtp_object_id
        JOIN sys.tables t on h.object_id=t.object_id
WHERE ia.type=1
ORDER BY [table], [index];
```

Demonstration

Monitoring chain length, and empty buckets

Memory-Optimized

Monitoring memory consumption



nory

ge by Memory

Memory-Optimized

Monitoring memory consumption

DEMO004A - Multi...RICA\sammes (88)) ~vsDAF8.sql - DES...ERICA\sammes (62)) ~vsD45E.sql - DES...ERICA\sammes (61))

```
SELECT
    object_name(object_id) AS Name
    ,*
FROM
    sys.dm_db_xtp_table_memory_stats

```

200 %

Results | Messages

	Name	object_id	memory_allocated_for_table_kb	memory_used_by_table_kb	memory_allocated_for_indexes_kb	memory_used_by_indexes_kb
1	Product_Workload	597577167	2688	2388	1760	1088
2	Product	629577281	512	112	192	7

Demonstration

Memory utilization

Memory-Optimized – lock free access

- Isolation Level is the DEFAULT (Read Committed

```sql
-- check session' ISOLATION LEVEL
SELECT transaction_isolation_level, *
FROM sys.dm_exec_sessions
WHERE session_id = @@SPID
```

200 %

Results | Messages

	transaction_isolation_level	session_id	login_time	host_name
1	2	88	2020-03-11 21:42:14.610	DESKTOP-ATUR

Memory-Optimized – lock free access

- Isolation Level is the DEFAULT (Read Committed)

Session 1

```sql
SELECT
    MakeFlag,
    *
FROM
    [Product]
WHERE
    ProductID = 904

BEGIN TRANSACTION
    SELECT @@TRANCOUNT
    UPDATE
        [Product] WITH (SNAPSHOT)
    SET
        MakeFlag = 0
    WHERE
        ProductID = 904
```

Session 2

Can another session access this row that is being updated?

	ProductNumber
r, 40	FR-M63S-40

```sql
WHERE
    ProductID >= 900



---- transaction / errors
```

200 %

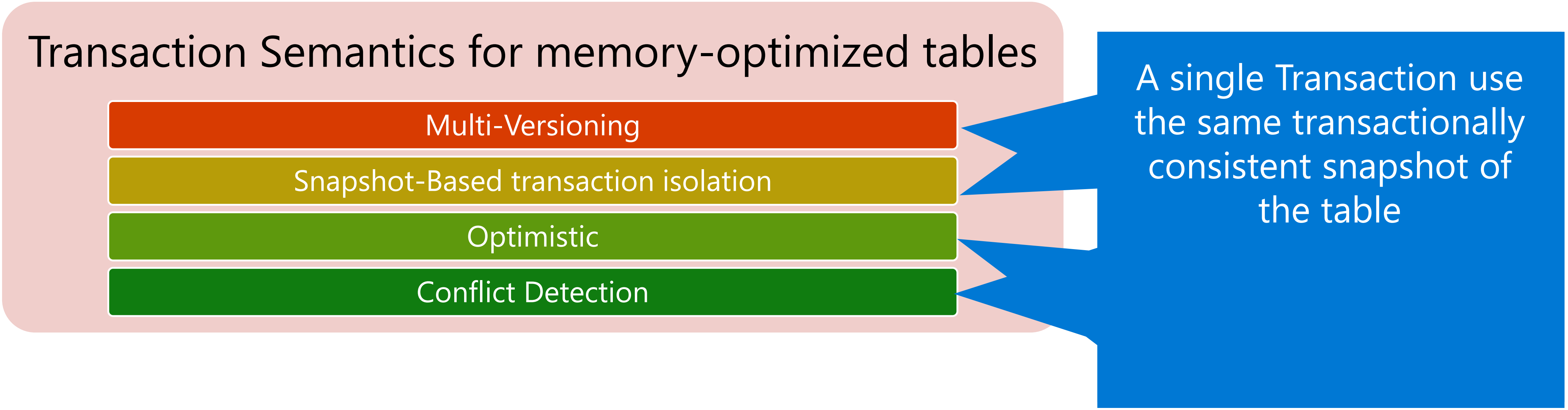
Results | Messages

	MakeFlag	ProductID	Name	ProductNumber
1	1	900	LL Touring Frame - Yellow, 50	FR-T67Y-50
2	1	901	LL Touring Frame - Yellow, 54	FR-T67Y-54
3	1	902	LL Touring Frame - Yellow, 58	FR-T67Y-58
4	1	903	LL Touring Frame - Blue, 44	FR-T67U-44
5	1	904	ML Mountain Frame-W - Silver, 40	FR-M63S-40
6	1	905	ML Mountain Frame-W - Silver, 42	FR-M63S-42

Transaction Handling

Memory-Optimized

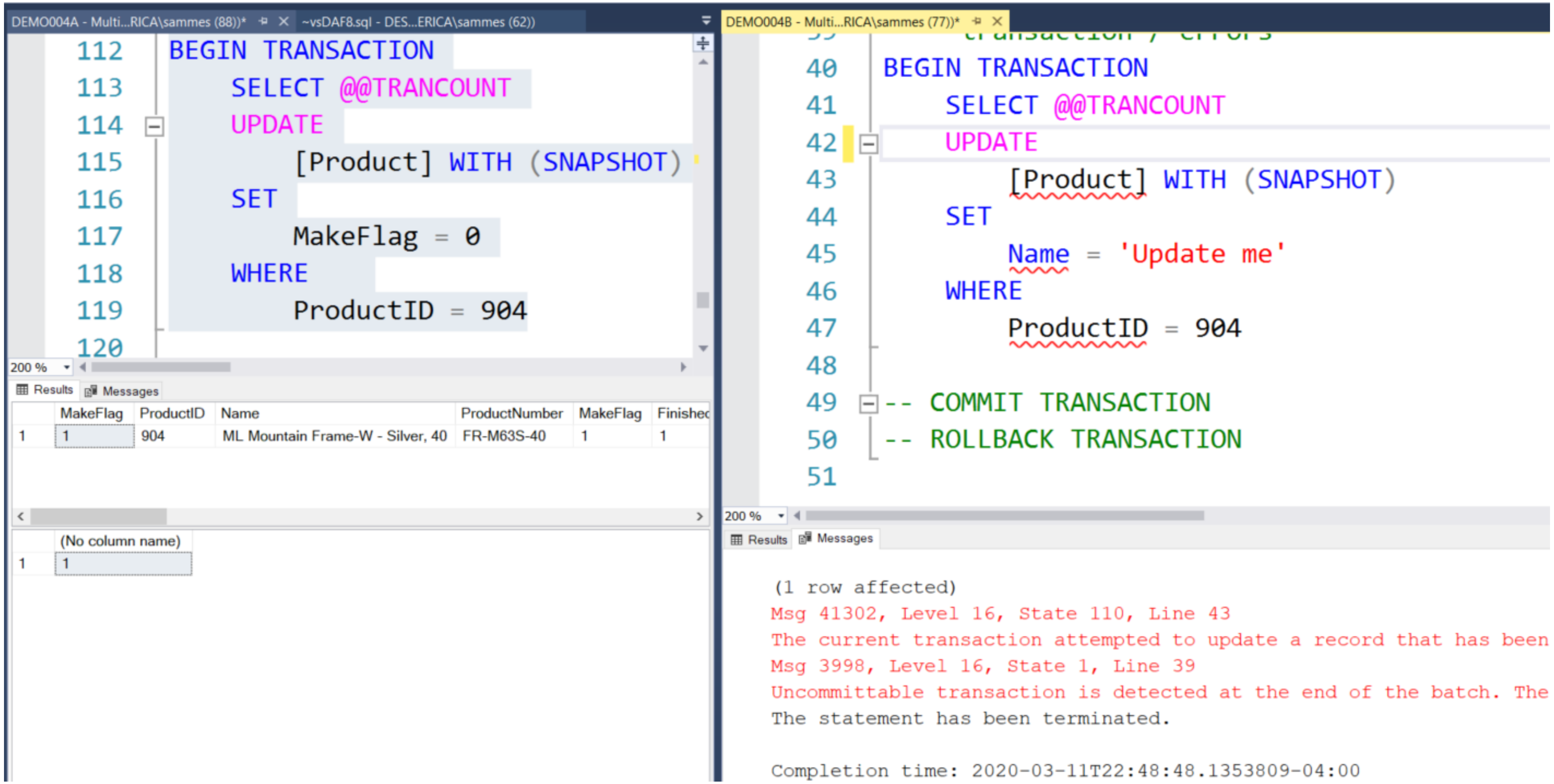
- Allow optimistic concurrency control with higher isolation level:
  - REPEATABLE READ
  - SERIALIZABLE
    No locks are taken to enforce isolation level
- A validation process is executed at the end of transactions
  If two transactions try to update same row a write/write, a conflict happens and it results in error (41302)



Transaction Handling

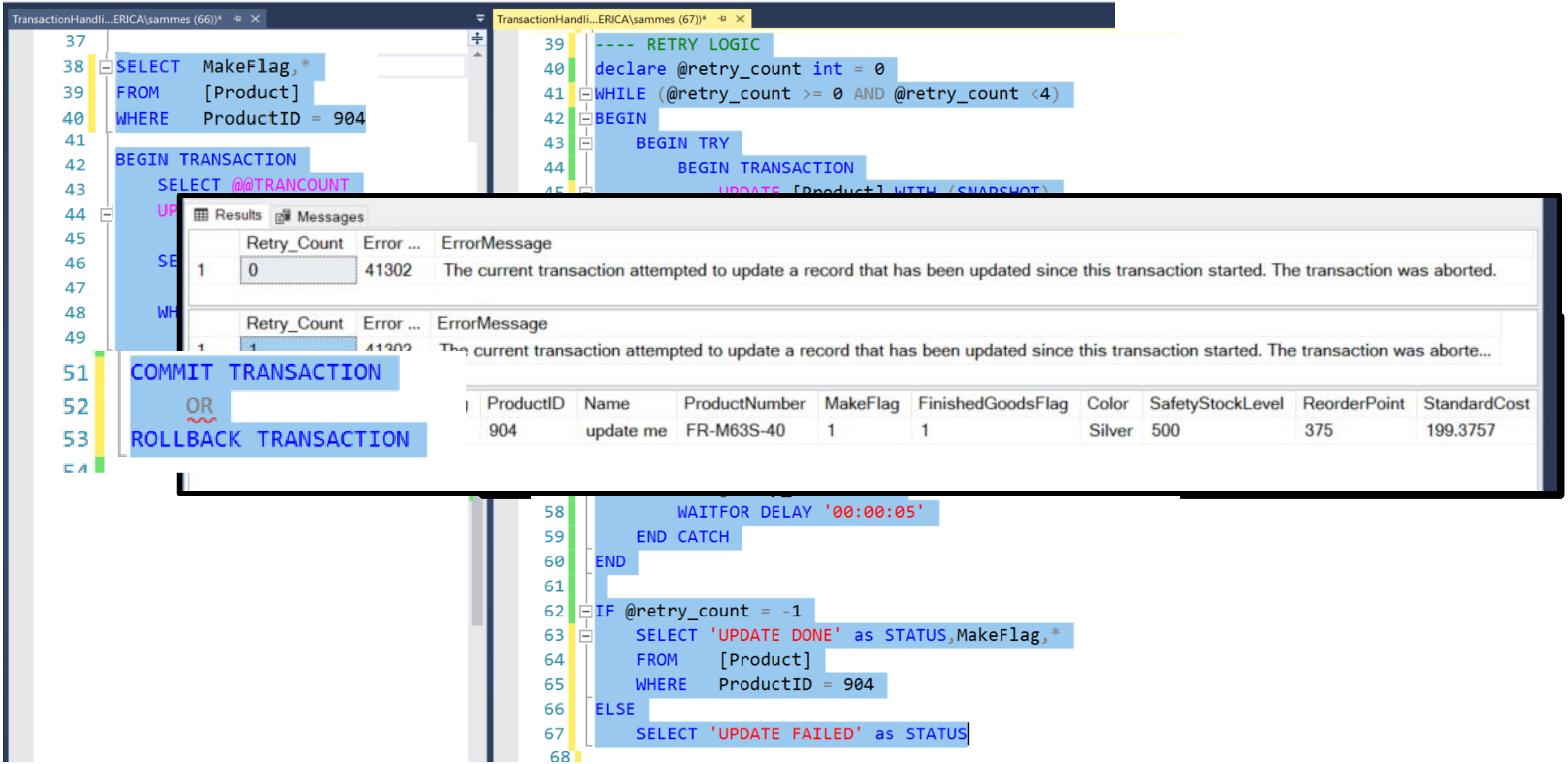
Without RETRY LOGIC

- Transaction Conflict handling



Transaction Handling

With RETRY LOGIC



Optional Demonstration

Concurrent Transactions & Retry Logic

Natively Compiled Stored Procedures

Native Compilation allows faster data access and more efficient query execution than Interpreted Transact-SQL (also called INTEROP)

The process converts T-SQL programming constructs into native code, which consists of processor instructions that won’t require further compilation or interpretation

Limitations with Native Compiled SPs:

Full Transact-SQL programmability is not implemented	Full Query Surface is not implemented	**SCHEMABINDING** is required	The procedure body must consist of exactly one **ATOMIC BLOCK** (with BEGIN ATOMIC)

Objects will get auto-recompilation at:

SQL Server restarts,	On database failover,	If database is taken offline and back online

Natively Compiled Stored Procedures

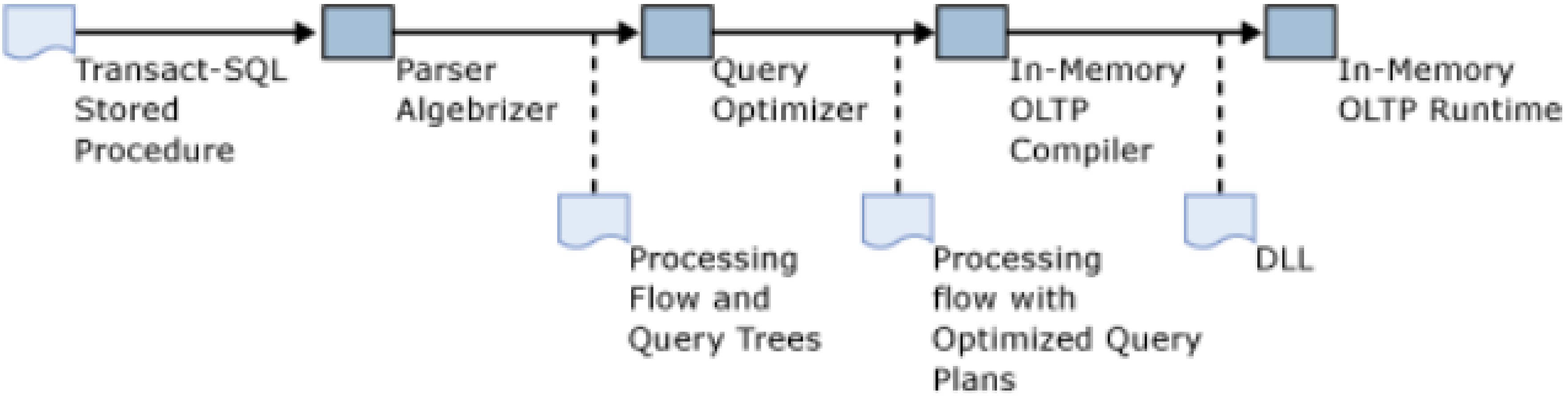
Comparison with Interpreted Stored Procedures

	Natively Compiled	**Interpreted**
Initial Compilation	At Create Time	At First Execution
Automatic Recompilation	Upon 1st execution after database or server restart	On Server restart, or if evicted from plan cache
Manual Recompilation	Use **sp_recompile**	Use **sp_recompile**
Parameter Sniffing	Doesn't apply. All parameters are considered to have **UNKNOWN** values	Use parameters from 1st execution to determine plan

Natively Compiled Stored Procedures

Compilation process

1. **CREATE PROCEDURE** is issued
2. Parser and Algebrizer create the processing flow.
3. Optimizer creates optimized query execution plans.
4. In-Memory OLTP compiler generates a DLL containing machine code
5. DLL is loaded into memory.



Memory-Optimized objects

Analysis tool

Wizard:

Analysis of existing tables and stored procedures to identify bottlenecks in the database.

Assistance to migrate these disk-based objects into Memory-Optimized.

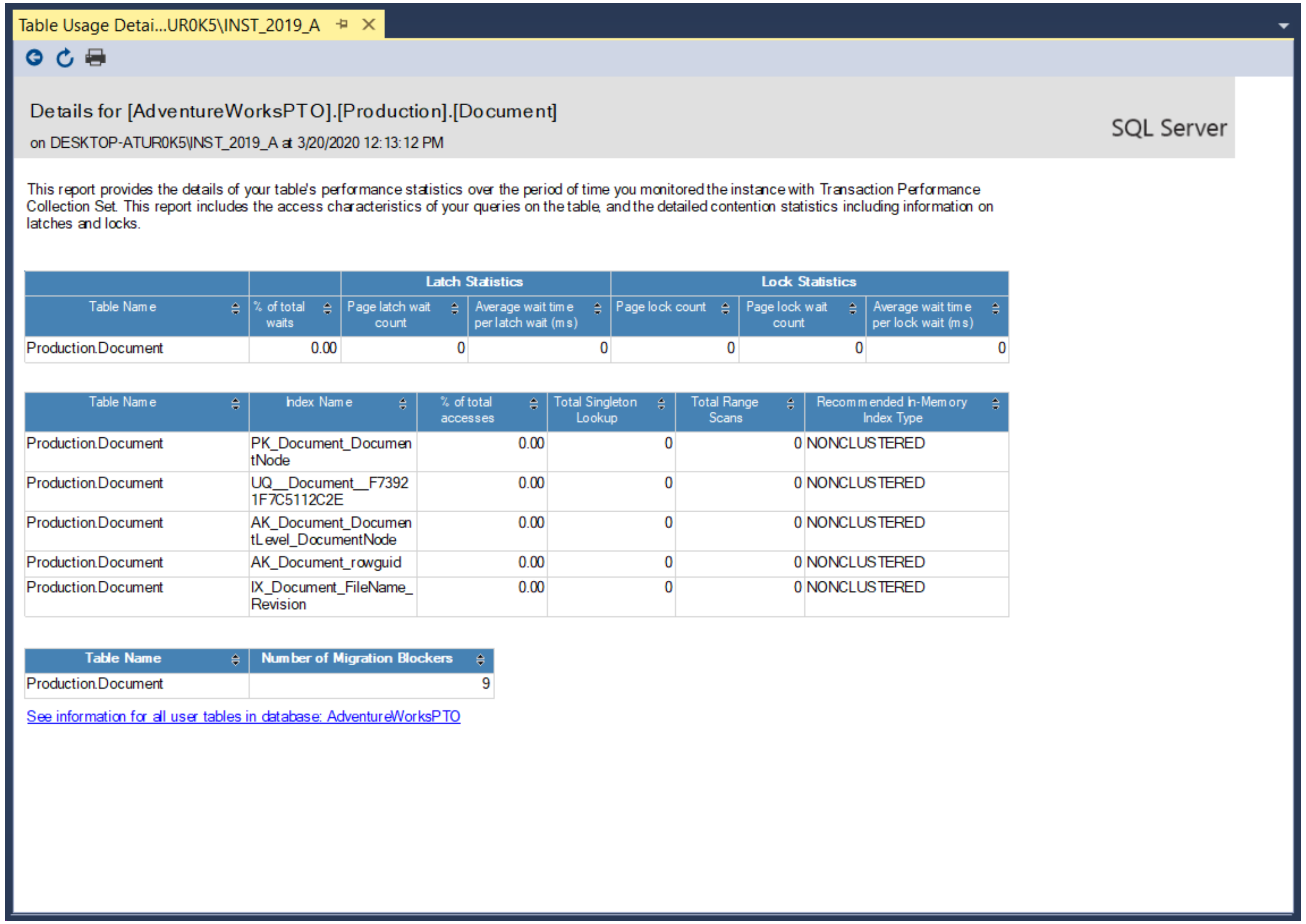


Table Usage Detai...UR0K5\INST_2019_A

Details for [AdventureWorksPTO].[Production].[Document]

on DESKTOP-ATUR0K5\INST_2019_A at 3/20/2020 12:13:12 PM

SQL Server

This report provides the details of your table's performance statistics over the period of time you monitored the instance with Transaction Performance Collection Set. This report includes the access characteristics of your queries on the table, and the detailed contention statistics including information on latches and locks.

		Latch Statistics		Lock Statistics		
Table Name	% of total waits	Page latch wait count	Average wait time per latch wait (ms)	Page lock count	Page lock wait count	Average wait time per lock wait (ms)
Production.Document	0.00	0	0	0	0	0

Table Name	Index Name	% of total accesses	Total Singleton Lookup	Total Range Scans	Recommended In-Memory Index Type
Production.Document	PK_Document_DocumentNode	0.00	0	0	NONCLUSTERED
Production.Document	UQ__Document__F73921F7C5112C2E	0.00	0	0	NONCLUSTERED
Production.Document	AK_Document_DocumentLevel_DocumentNode	0.00	0	0	NONCLUSTERED
Production.Document	AK_Document_rowguid	0.00	0	0	NONCLUSTERED
Production.Document	IX_Document_FileName_Revision	0.00	0	0	NONCLUSTERED

Table Name	Number of Migration Blockers
Production.Document	9

See information for all user tables in database: AdventureWorksPTO

Optional Demonstration

Natively compiled Stored
Procedure

Exploring In-memory OLTP Performance

- Comparing performance between Disk-Based and Memory-Optimized Tables

Questions?

Knowledge Check

What is the primary storage for Memory-Optimized Tables?

True or False: The index contents for Memory-Optimized Tables stored in the data file

What is the impact of having too many indexes on a Memory-Optimized Table?

What impact can possibly happen with a HASH index with a small bucket count?

What is the memory impact on having a HASH index with a large bucket count?

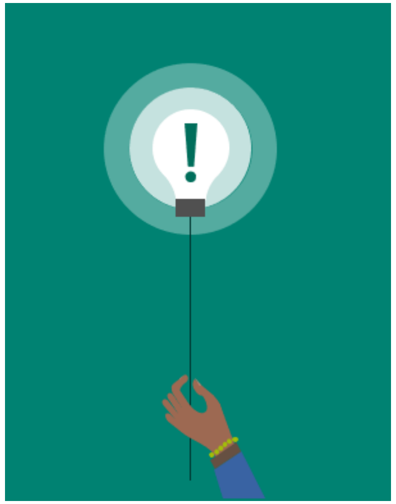
Are there performance advantages of having a non-chatty Natively Stored Procedure?

Extra: SQL Server Intelligent Query Processing

Objectives

After completing this learning, you will be able to:

- Understand the Intelligent query processing features.
- Enable/disable Intelligent query processing features.



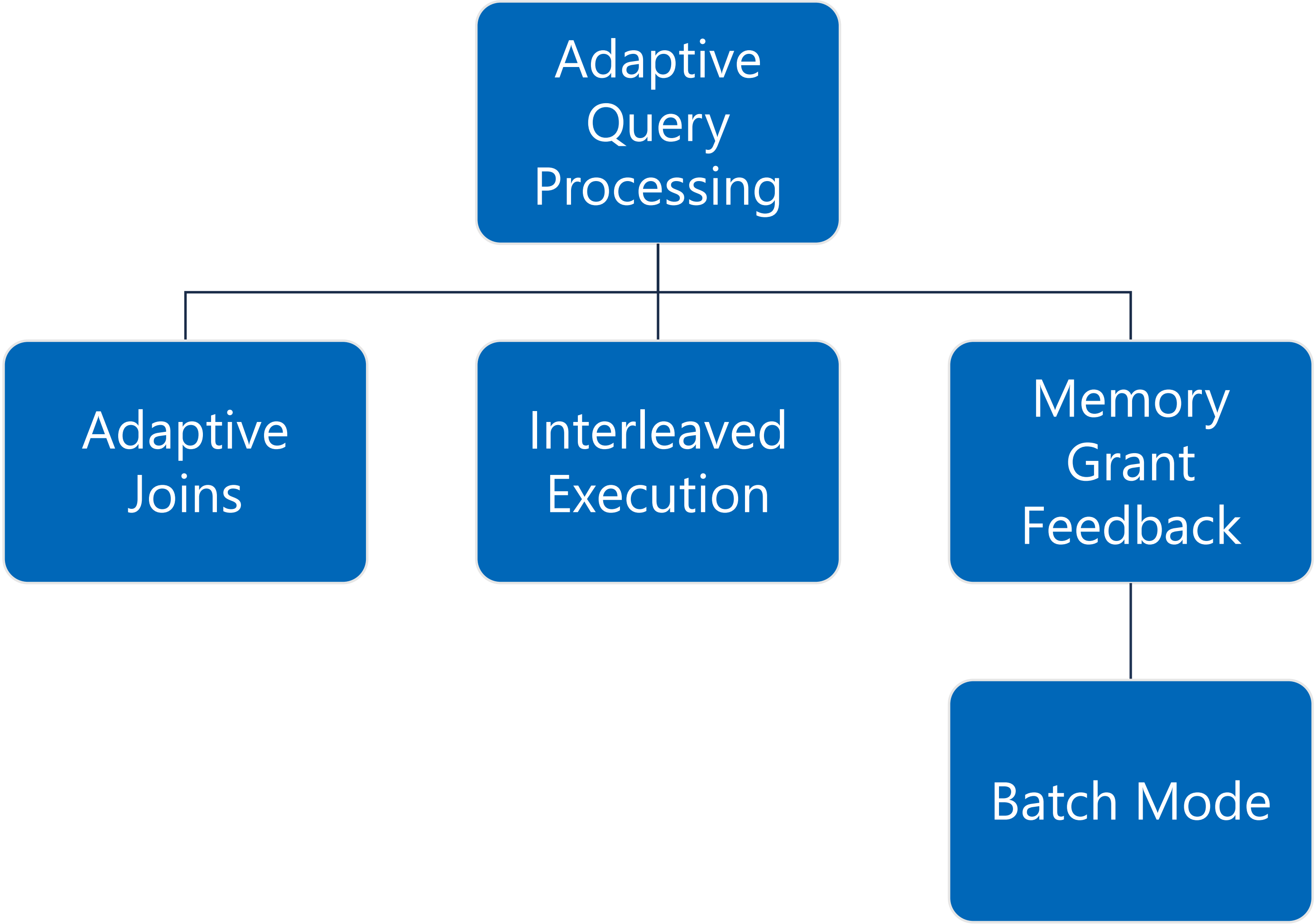
A History of Intelligent Query Processing

Adaptive Query Processing (2017)

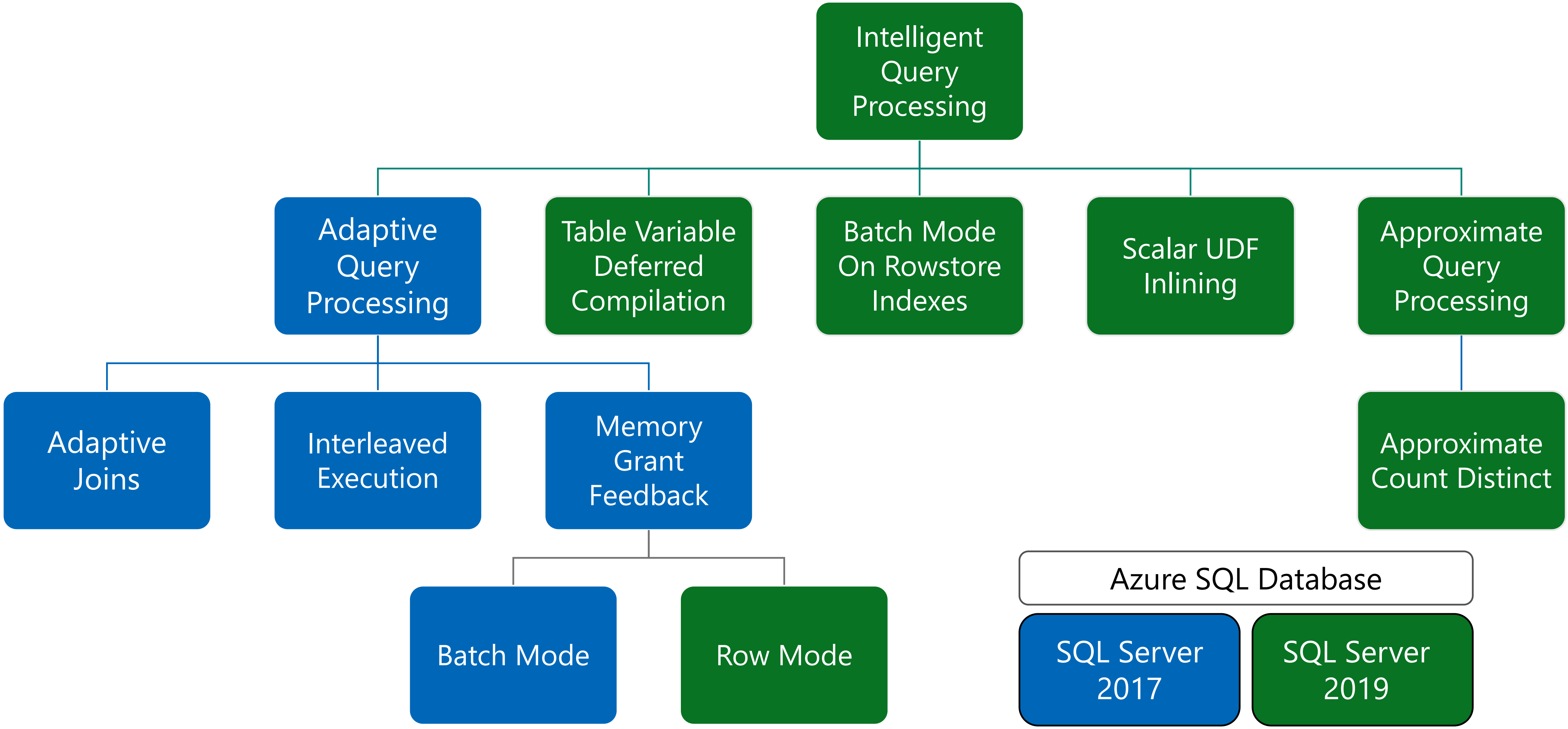
Intelligent Query Processing (2019)

New Features of IQP (2022)

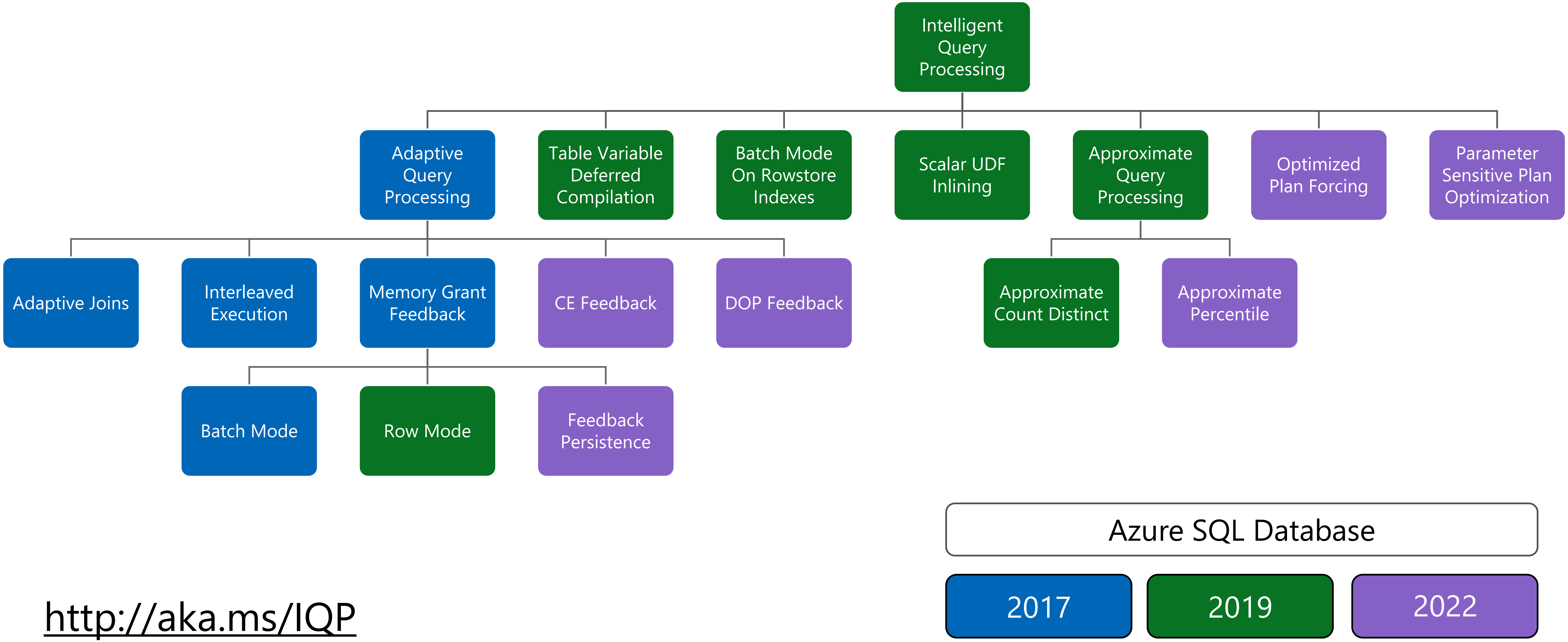
Adaptive Query Processing (2017)



Intelligent Query Processing (2019)



Intelligent Query Processing (2022)



http://aka.ms/IQP

Enabling and Disabling – Instance Level

For SQL Server 2017 Features

- Enabled by default in Compatibility level 140 or higher
- To disable change compatibility level to 130 or lower

For SQL Server 2019 Features

- Enabled by default in Compatibility level 150 or higher
- To disable change compatibility level to 140 or lower

For SQL Server 2022 Features

- Enabled by default in Compatibility level 160 or higher
- To disable change compatibility level to 150 or lower

Enabling and Disabling – Database Level

Different settings for 2017 vs Azure SQL, SQL Server 2019 and higher

```
ALTER DATABASE SCOPED CONFIGURATION SET DISABLE_BATCH_MODE_ADAPTIVE_JOINS = ON|OFF;
```

```
ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ADAPTIVE_JOINS = ON|OFF;
```

To get a list of Database Scoped Configuration settings

```
SELECT * From sys.database_scoped_configurations;
```

configuration_id	name	value
7	INTERLEAVED_EXECUTION_TVF	1
8	BATCH_MODE_MEMORY_GRANT_FEEDBACK	1
9	BATCH_MODE_ADAPTIVE_JOINS	1
10	TSQL_SCALAR_UDF_INLINING	1
16	ROW_MODE_MEMORY_GRANT_FEEDBACK	1
18	BATCH_MODE_ON_ROWSTORE	1
19	DEFERRED_COMPILATION_TV	1
28	PARAMETER_SENSITIVE_PLAN_OPTIMIZATION	1
31	CE_FEEDBACK	1
33	MEMORY_GRANT_FEEDBACK_PERSISTENCE	1
34	MEMORY_GRANT_FEEDBACK_PERCENTILE_GRANT	1
35	OPTIMIZED_PLAN_FORCING	0

Enabling and Disabling – Statement Level

You can disable features at the statement scope if necessary.

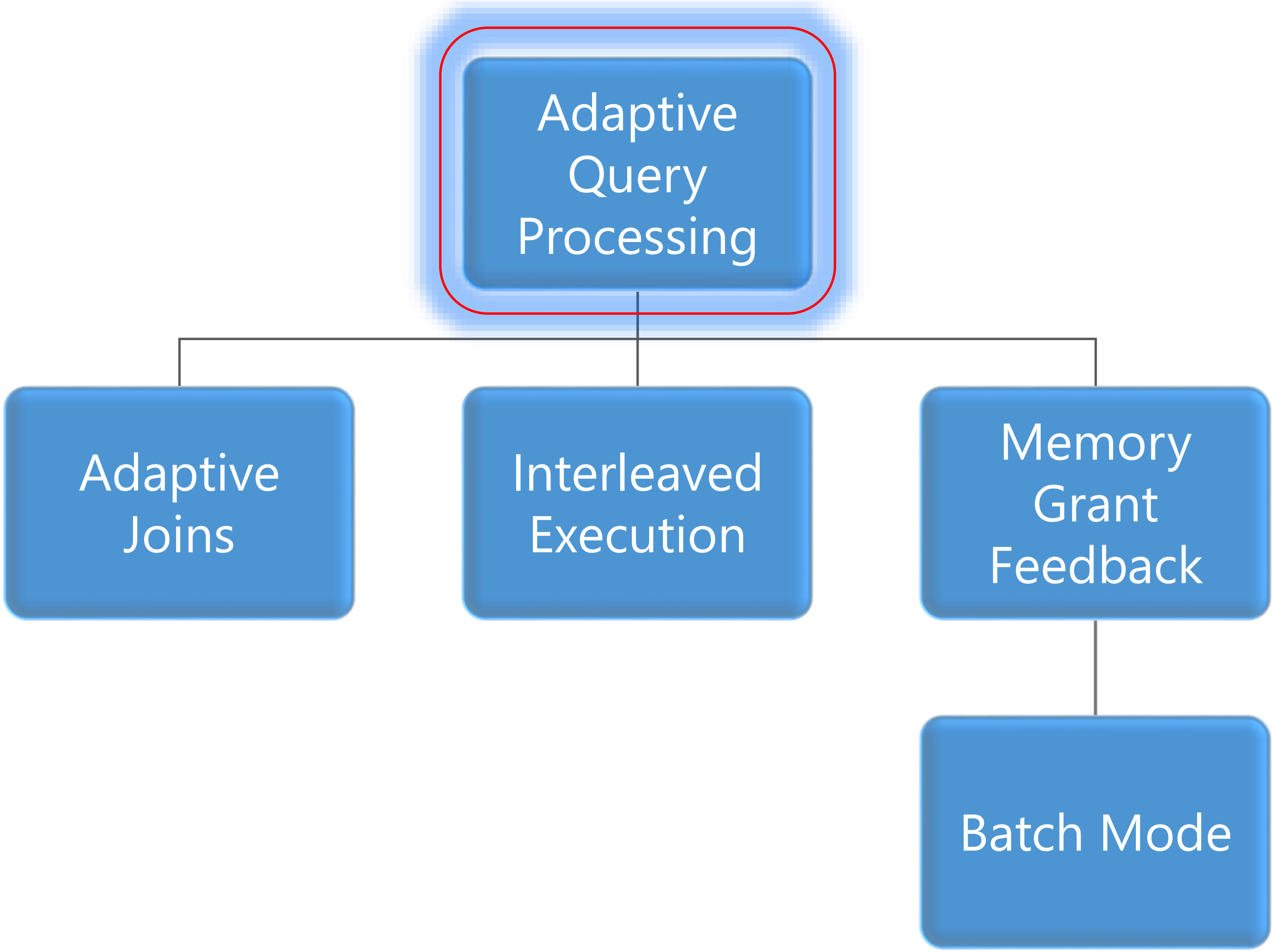
```
<statement>
OPTION (USE HINT('DISABLE_BATCH_MODE_ADAPTIVE_JOINS'));
```

To get a list of valid query use hints

```
SELECT * FROM sys.dm_exec_valid_use_hints;
```

name
DISABLE_INTERLEAVED_EXECUTION_TVF
DISABLE_BATCH_MODE_MEMORY_GRANT_FEEDBACK
DISABLE_BATCH_MODE_ADAPTIVE_JOINS
DISABLE_ROW_MODE_MEMORY_GRANT_FEEDBACK
DISABLE_DEFERRED_COMPILATION_TV
DISABLE_TSQL_SCALAR_UDF_INLINING
ASSUME_FULL_INDEPENDENCE_FOR_FILTER_ESTIMATES
ASSUME_PARTIAL_CORRELATION_FOR_FILTER_ESTIMATES
DISABLE_CE_FEEDBACK
DISABLE_MEMORY_GRANT_FEEDBACK_PERSISTENCE
DISABLE_DOP_FEEDBACK
DISABLE_OPTIMIZED_PLAN_FORCING

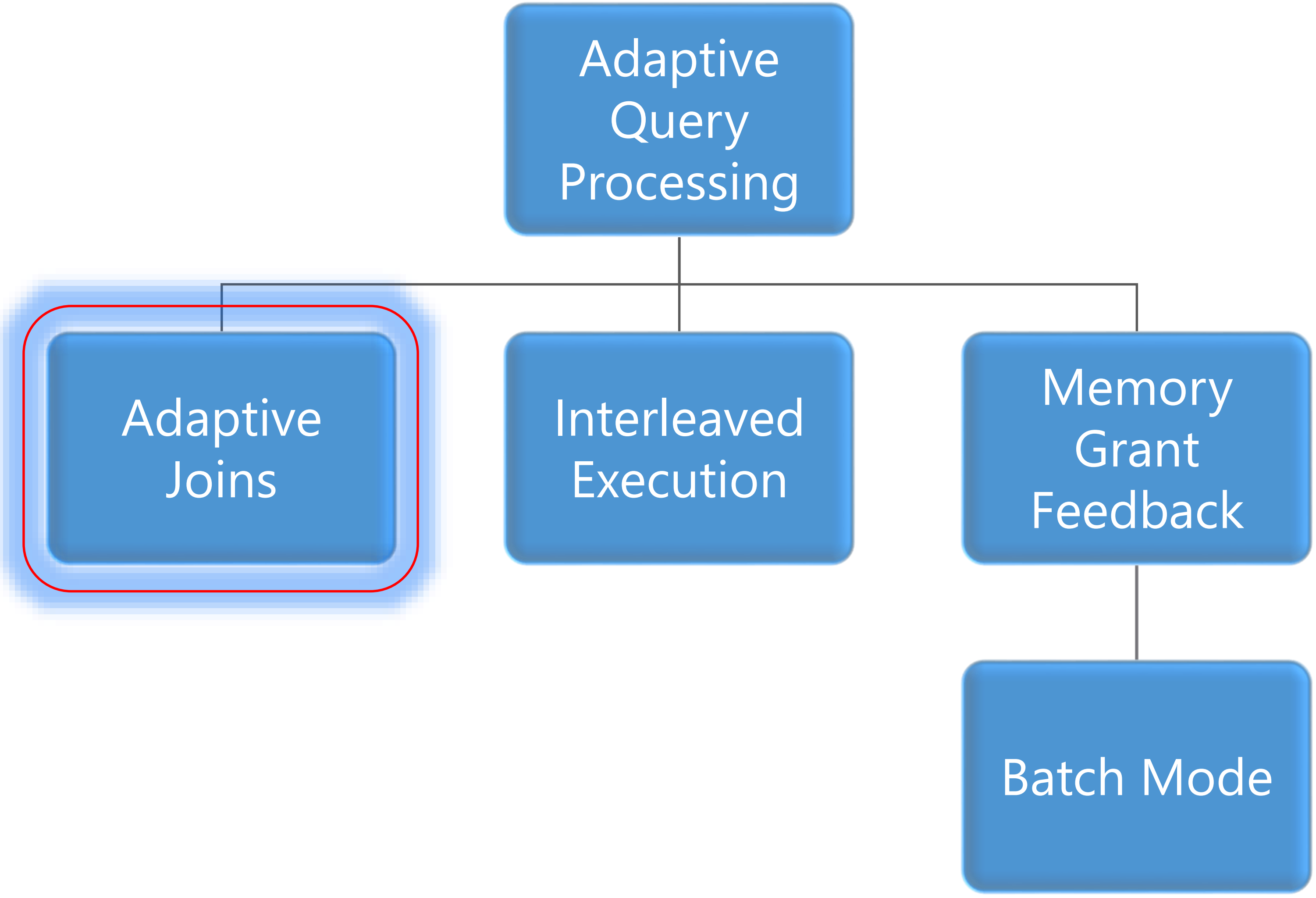
Adaptive Query Processing (2017)



Addresses performance issues related to the cardinality estimation of an execution plan.

These options can provide improved join type selection, row-calculations for Multi-Statement Table-Valued Functions, and memory allocation of row storage.

Batch Mode Adaptive Joins (2017)



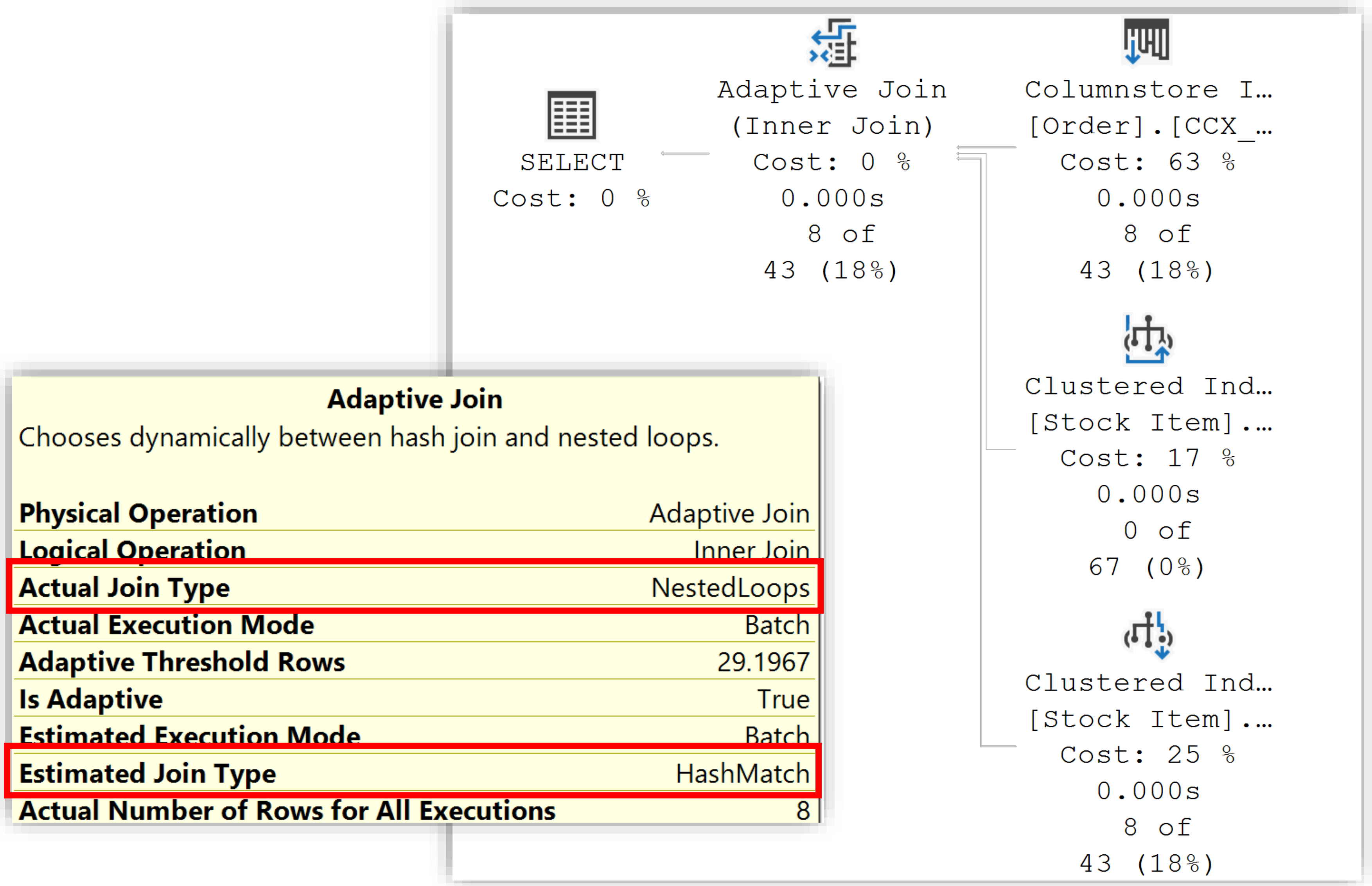
This feature enables the choice of either the Hash or the Nested Loop join type.

Decision is deferred until statement execution.

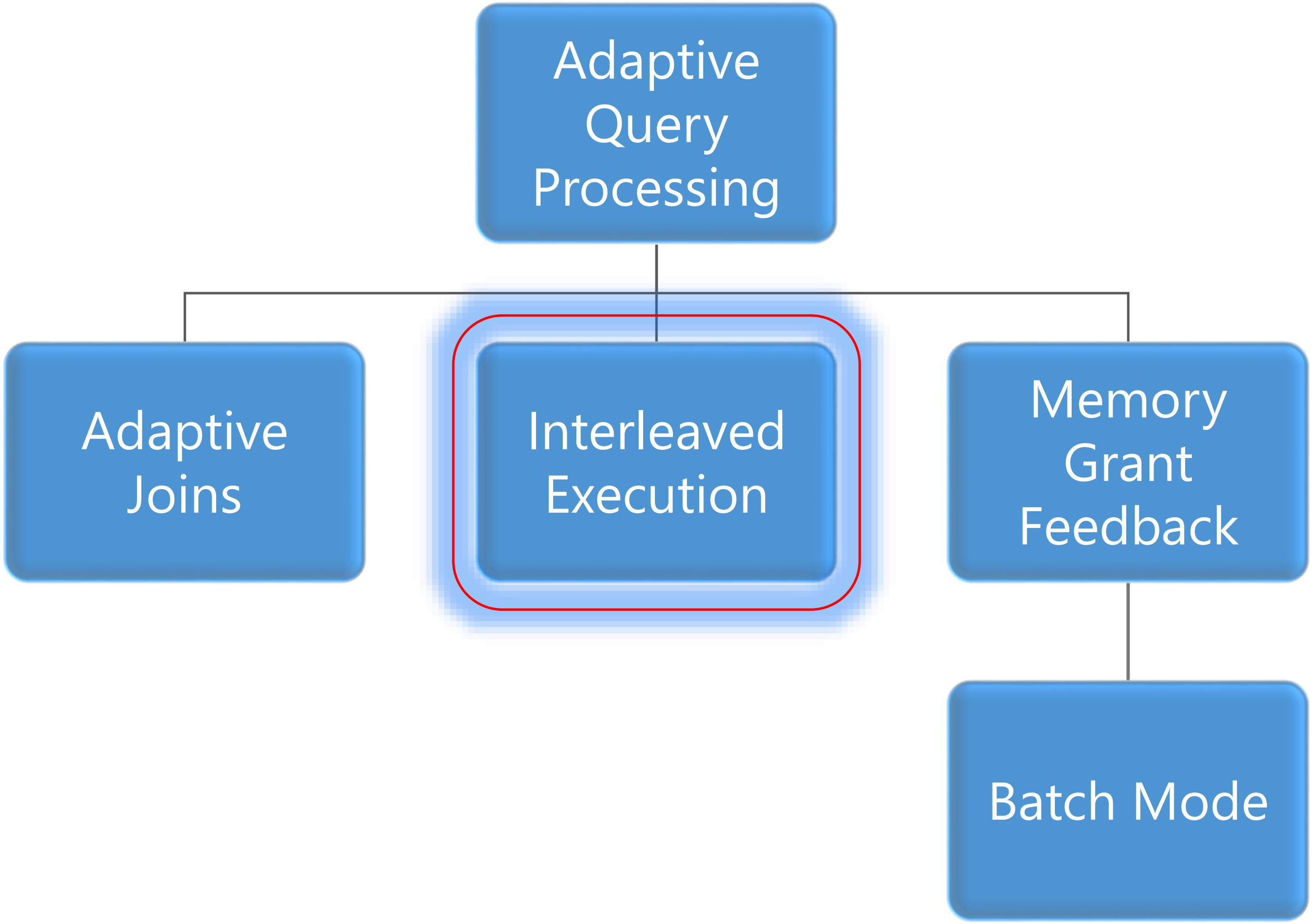
No need to use join hints in queries.

Batch Mode Adaptive Joins (2017)

Adaptive Joins



Interleaved Execution (2017)



Previously, when a Multi-Statement Table-Valued Function was executed, it used a fixed row estimate of 100 rows.

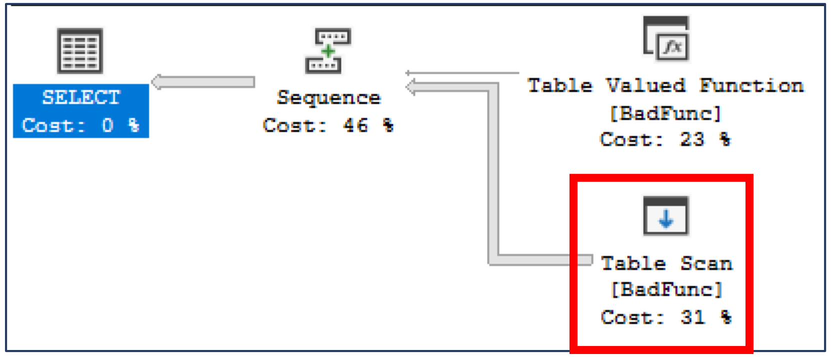
Now execution is paused so a better cardinality estimate can be captured.

Interleaved Execution (2017)

Interleaved Execution

Compatibility Level 120/130

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	12345
Actual Number of Rows	12345
Actual Number of Batches	0
Estimated Operator Cost	0.003392 (92%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.000267
Estimated Subtree Cost	0.003392
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows to be Read	100
Estimated Number of Rows	100
Estimated Row Size	67 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	2

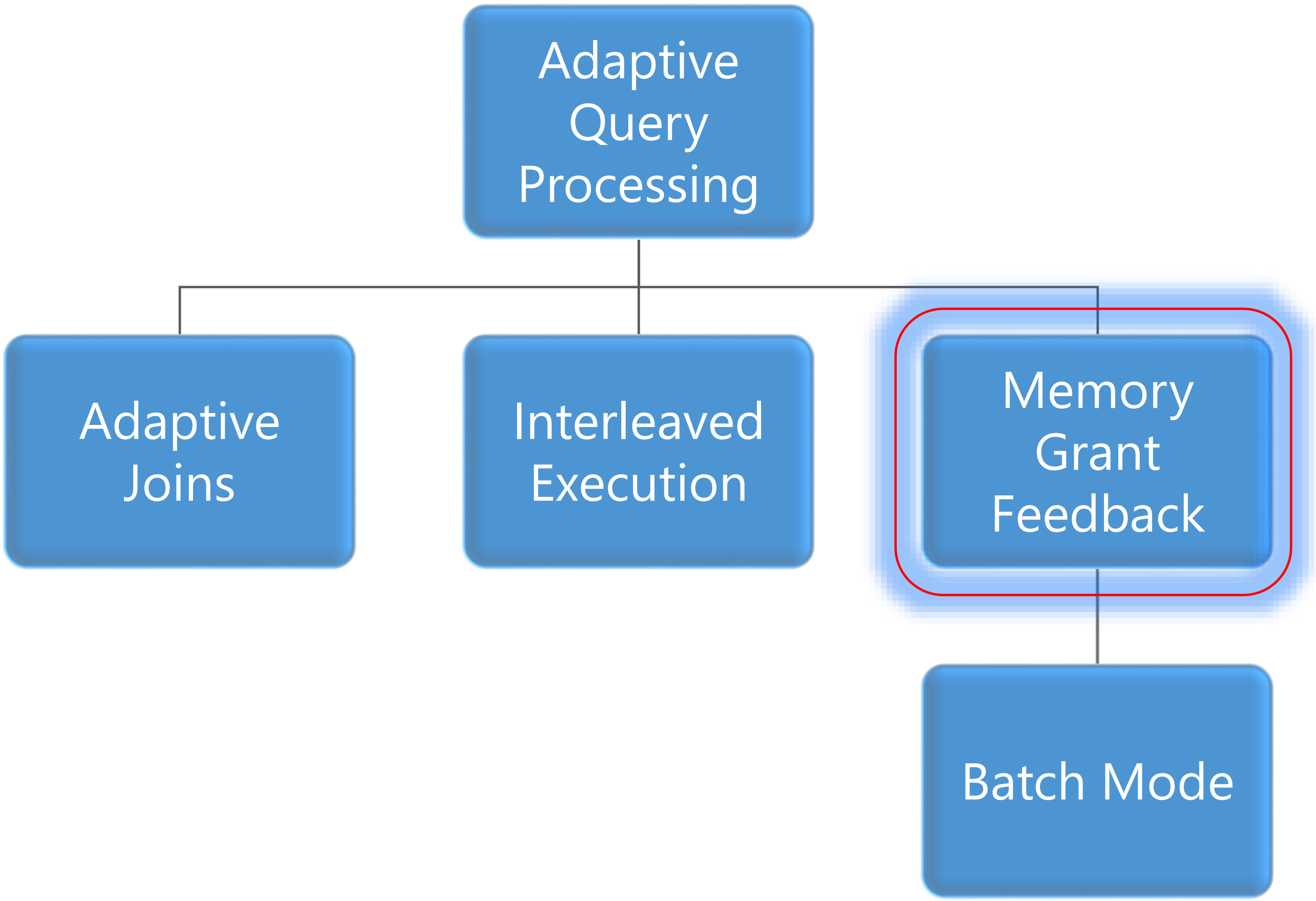


During optimization if SQL Server encounter a read-only multi-statement table-valued function (MSTVF), it will pause optimization, execute the applicable subtree, capture accurate cardinality estimates, and then resume optimization for downstream operations.

Compatibility Level 140 or higher

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	12345
Actual Number of Rows	12345
Actual Number of Batches	0
Estimated Operator Cost	0.0168615 (31%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0137365
Estimated Subtree Cost	0.0168615
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows to be Read	12345
Estimated Number of Rows	12345
Estimated Row Size	67 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	2

Batch Mode Memory Grant Feedback (2017)



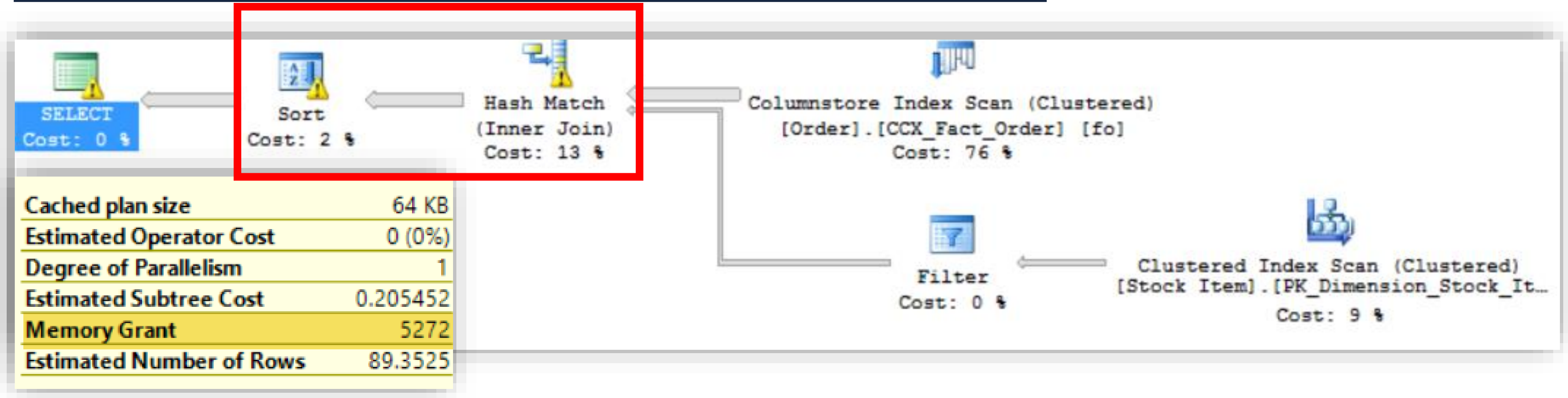
When compiling an execution plan, the query engine estimates how much memory is needed to store rows during join and sort operations.

Too much memory allocation may impact performance of other operations. Not enough will cause a spill over to disk.

This feature recalculates memory on first execution and updates the cached plan.

Batch Mode Memory Grant Feedback (2017)

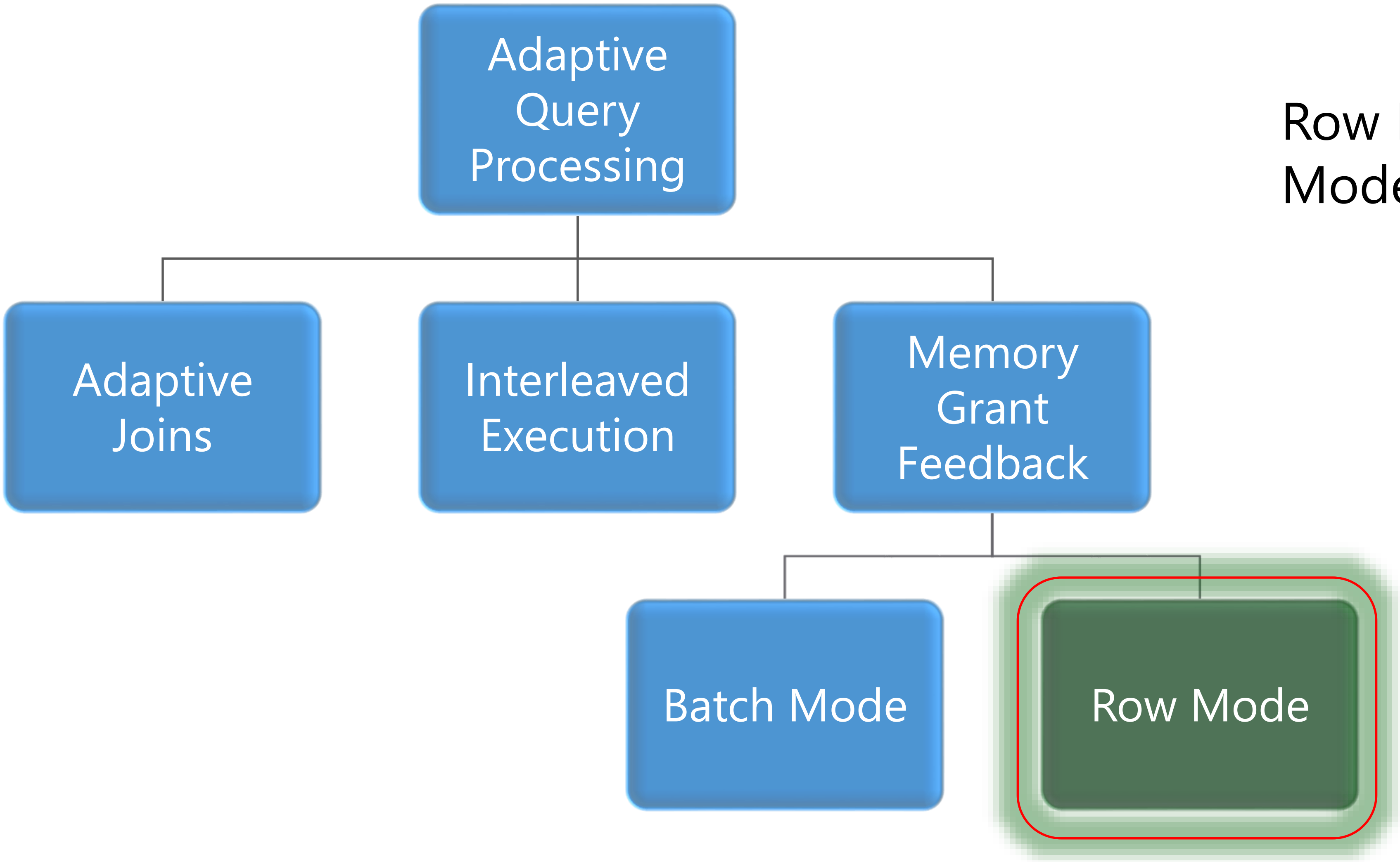
First Execution (Spills detected; feedback generated)



Second Execution (Memory grant adjusted)

Memory Grant Feedback (Batch Mode)

Row Mode Memory Grant Feedback (2019)



Row Mode is just like Batch Mode, but different.

Row Mode Memory Grant Feedback (2019)

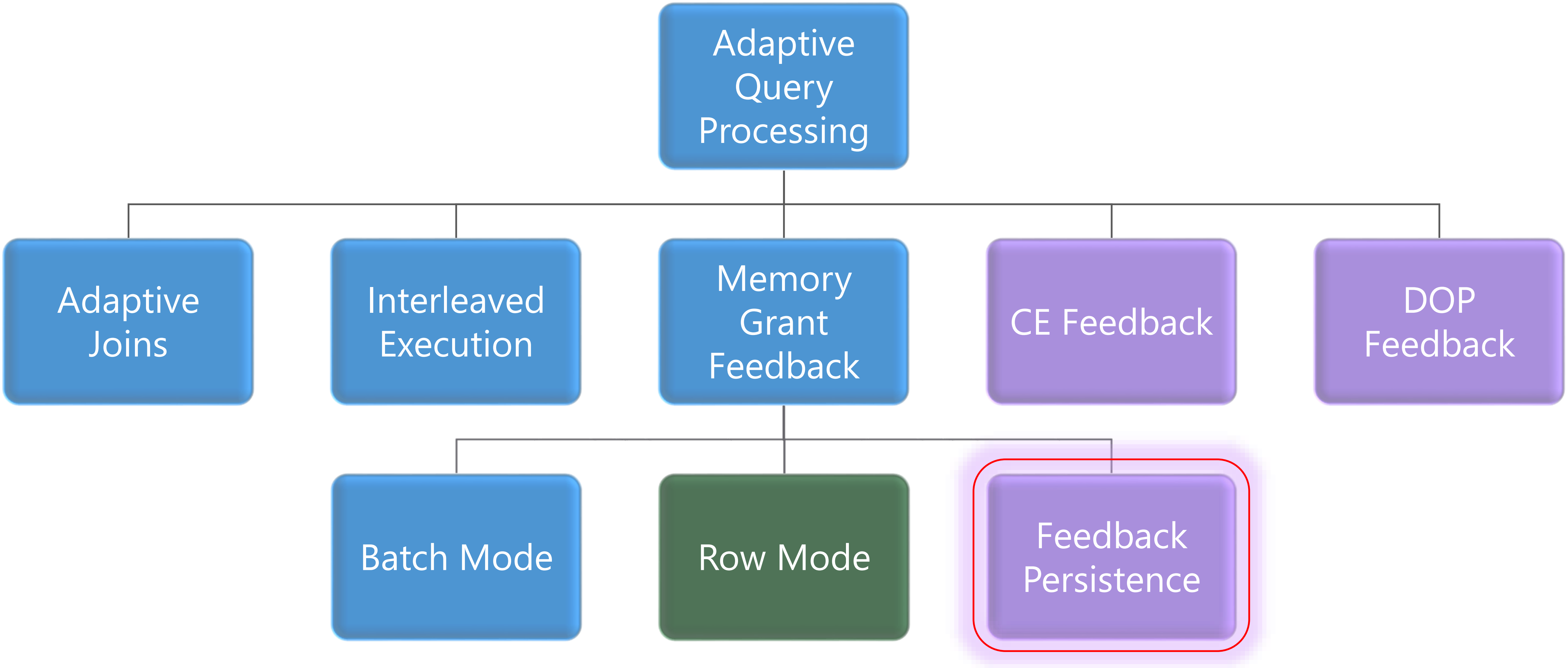
Expands on the batch mode memory grant feedback feature by also adjusting memory grant sizes for row mode operators.

MemoryGrantInfo	
DesiredMemory	13992
GrantedMemory	13992
GrantWaitTime	0
IsMemoryGrantFeedbackAdjusted	YesStable
LastRequestedMemory	13992
MaxQueryMemory	1497128
MaxUsedMemory	3744

Memory Grant Feedback (Row Mode)

Two new query plan attributes will be shown for actual post-execution plans.

Feedback Persistence (2022)



Feedback Persistence and Percentile (2022)

Problem: Cache Eviction

- Feedback is not persisted if the plan is evicted from cache or failover
- Record of how to adjust memory is lost and must re-learn

Solution: Persist the feedback

- Persist the memory grant feedback in the Query Store

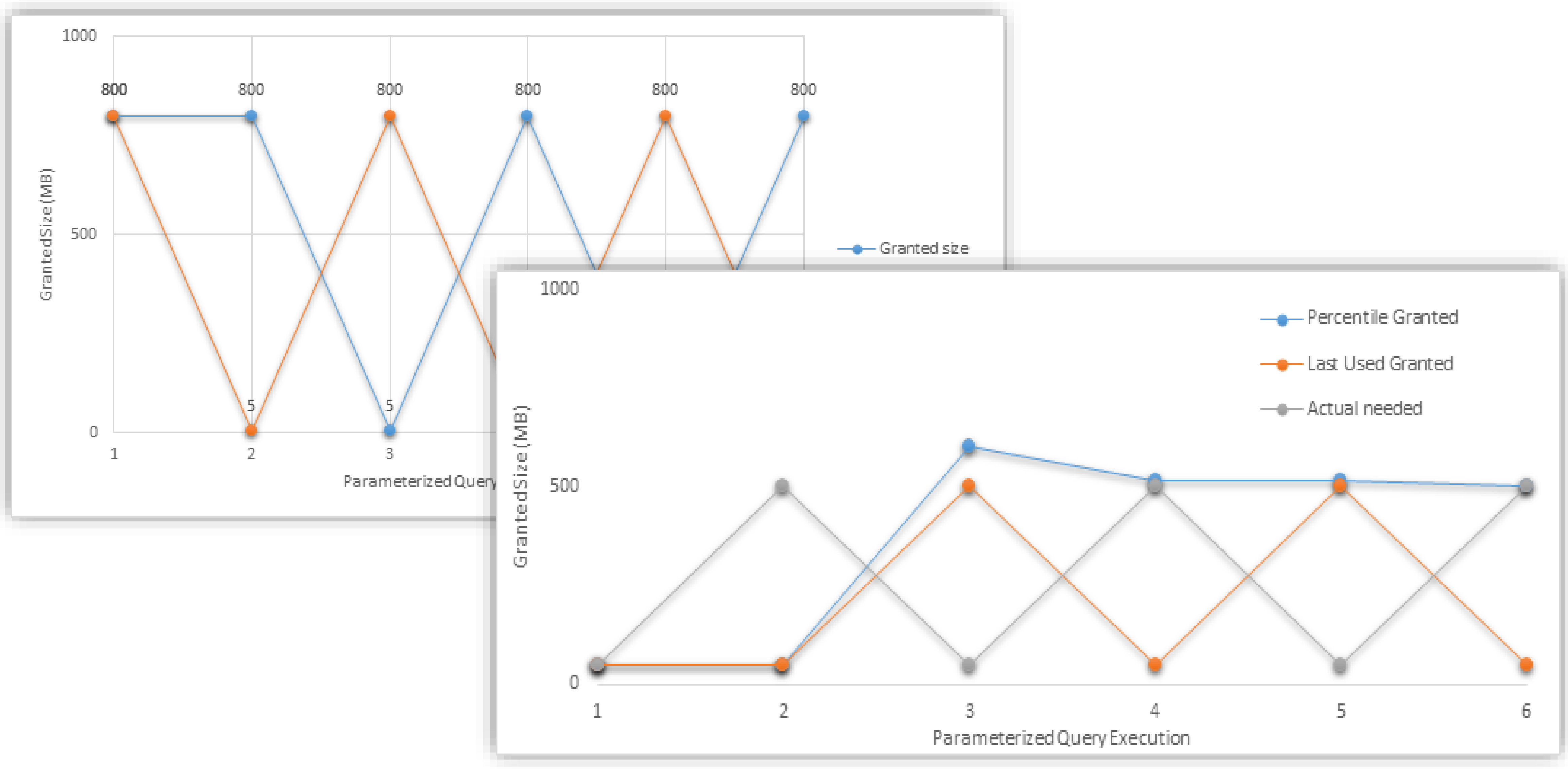
Problem: Oscillating Feedback

- Memory grants adjusted based on last feedback
- Parameter Sensitive Plans could change feedback

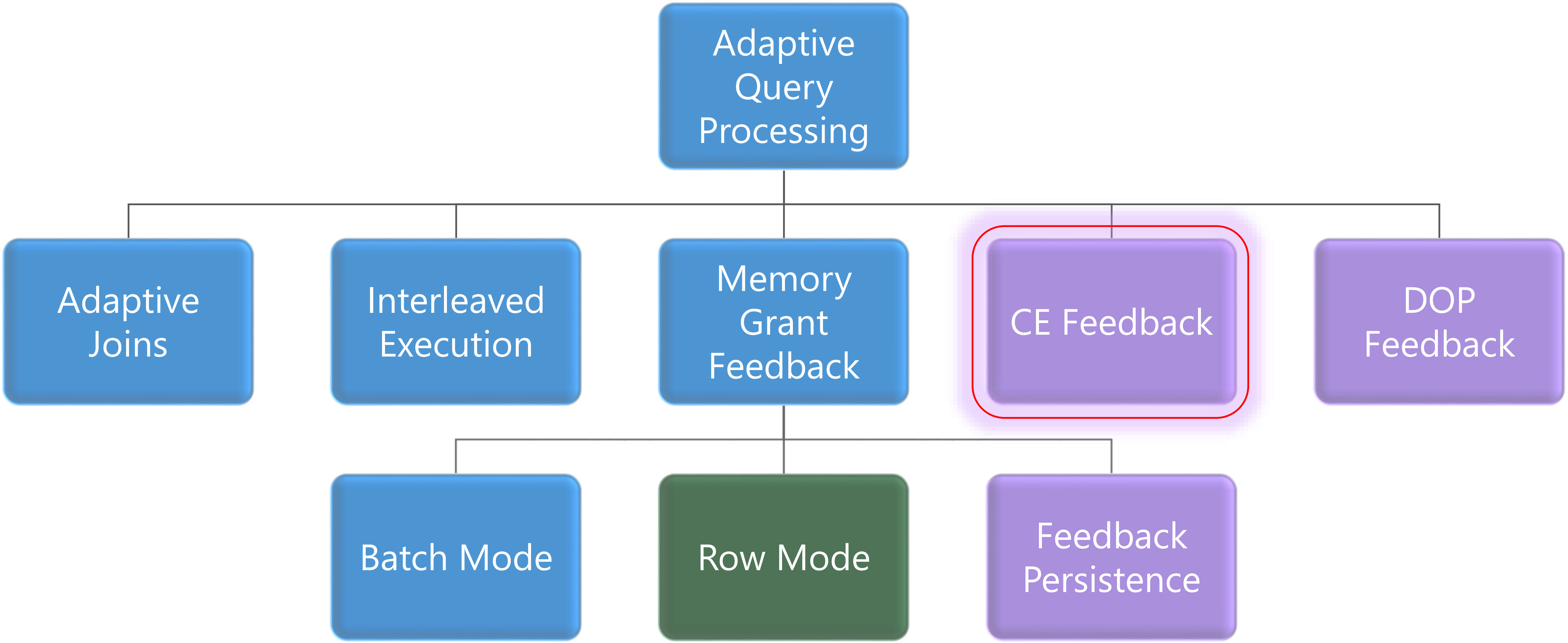
Solution: Percentile-based calculation

- Smooths the grant size values based on execution usage history

Feedback Persistence and Percentile (2022)



Cardinality Estimator Feedback (2022)



Cardinality Estimator Feedback (2022)

Cardinality Estimation Today

- CE determines the estimated number of rows for a query plan
- CE models are based on statistics and assumptions about the distribution of data
- Learn more about CE models and assumptions https://aka.ms/sqlCE

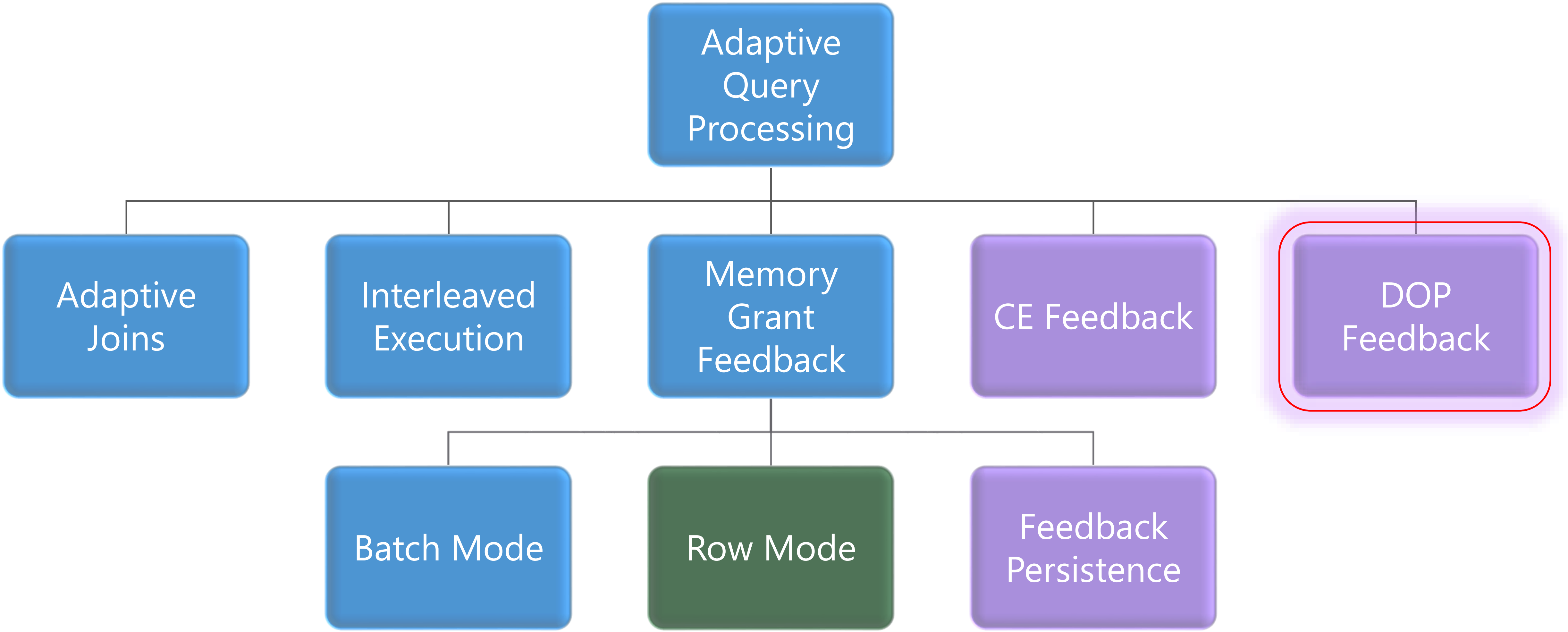
Problem: Incorrect Assumptions for Cardinality Estimates

- The cardinality estimator sometimes makes incorrect assumptions
- Poor assumptions leads to poor query plans.
- One CE models doesn’t fit all scenarios

Solution: Learn from historical CE model assumptions

- CE Feedback will evaluate accuracy for repeated queries
- If assumption looks incorrect, test a different CE model assumption and verify if it helps
- If a CE model assumption does help, it will replace the current plan in cache.

Degree of Parallelism Feedback (2022)



Degree of Parallelism Feedback (2022)

Parallelism Today

- Parallelism is often beneficial for querying large amounts of data, but transactional queries could suffer when time spent coordinating threads outweighs the advantages of using a parallel plan

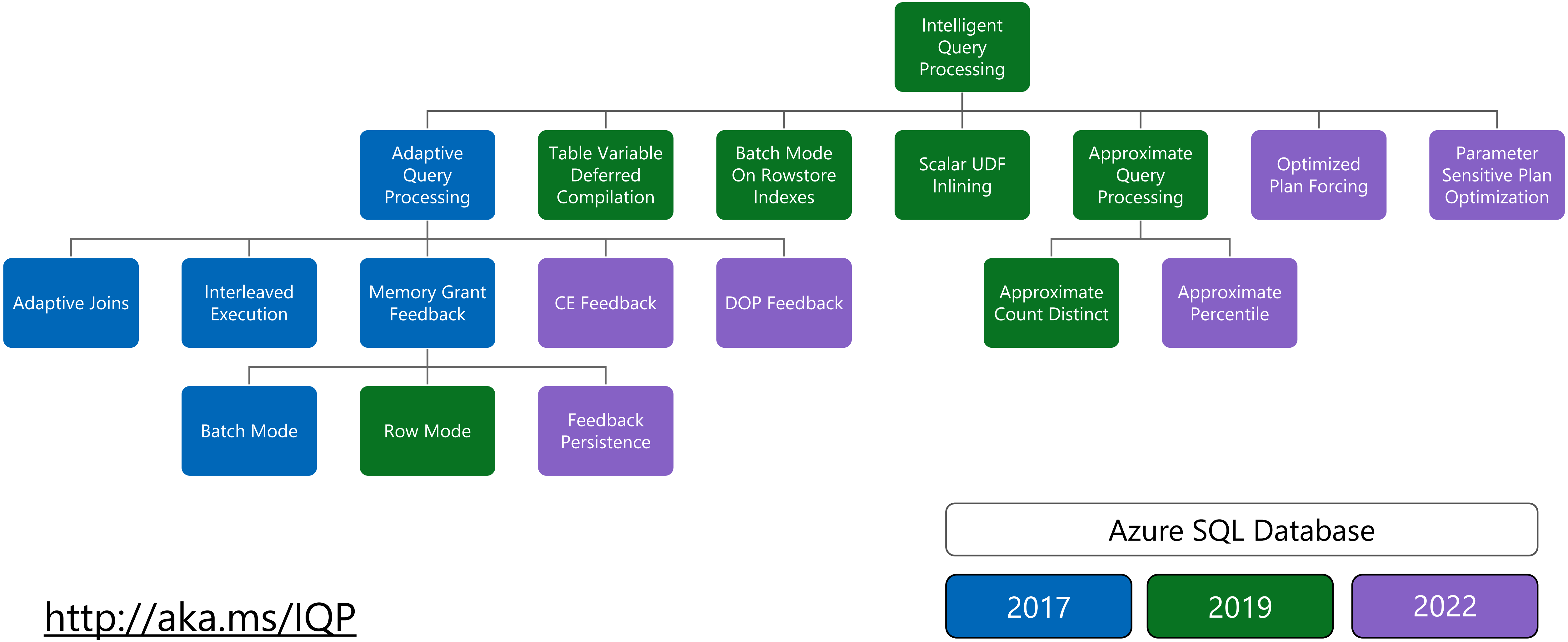
Current Settings

- Before SQL Server 2019, default value for MAXDOP = 0
- With SQL Server 2019, default is calculated at setup based on available processors
- Azure SQL Database the default MAXDOP is 8

DOP Feedback

- DOP Feedback will **identify** parallelism inefficiencies for repeating queries, based on CPU time, elapsed time, and waits
- If parallelism usage is inefficient, the DOP will be **lowered** for next execution (min DOP = 2) and then **verify** if it helps
- Only verified feedback is persisted (Query Store).
  - If next execution regresses, back to last good known DOP

Intelligent Query Processing (2022)



http://aka.ms/IQP

Intelligent Query Processing (2019 Features)

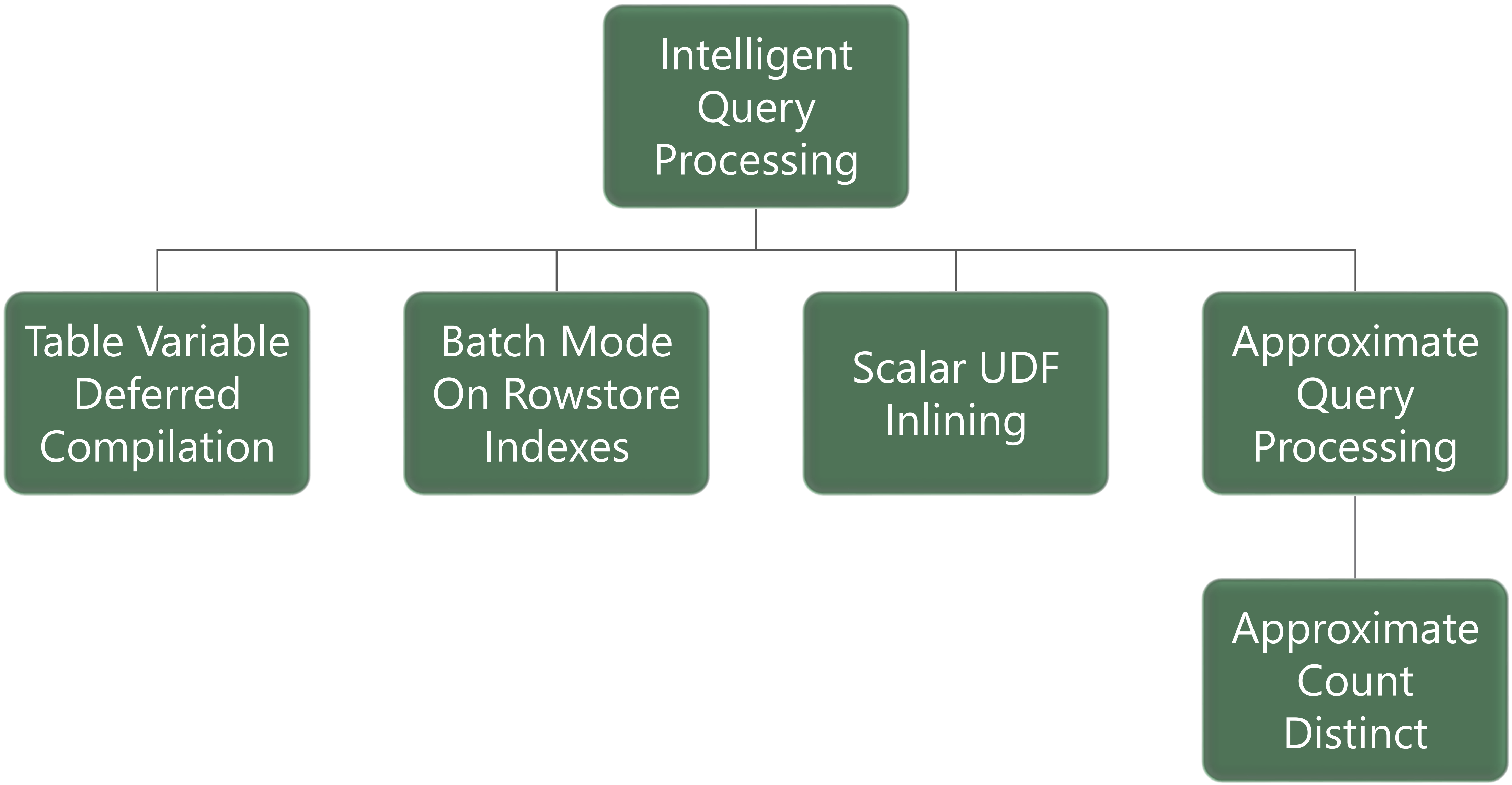
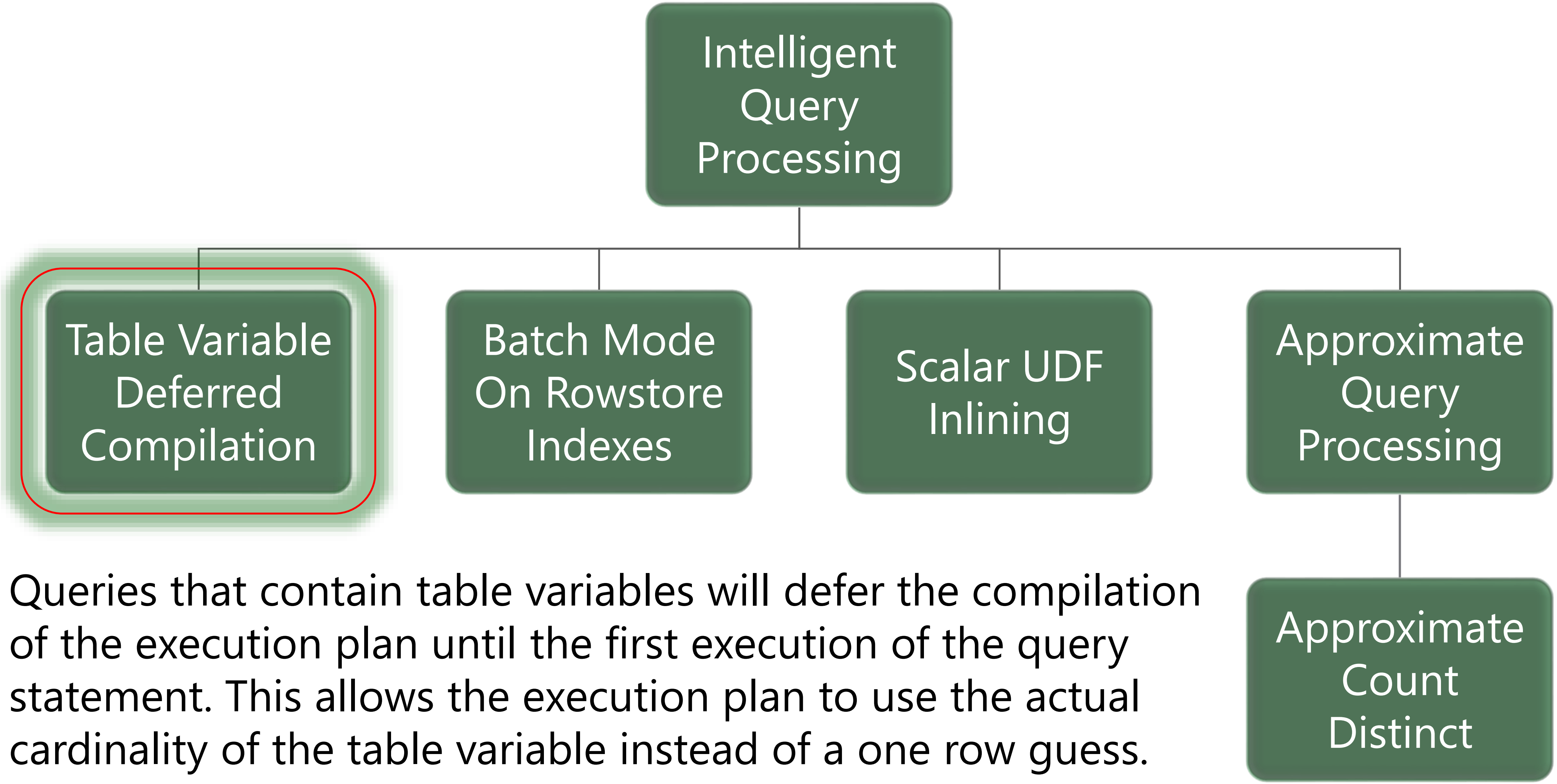
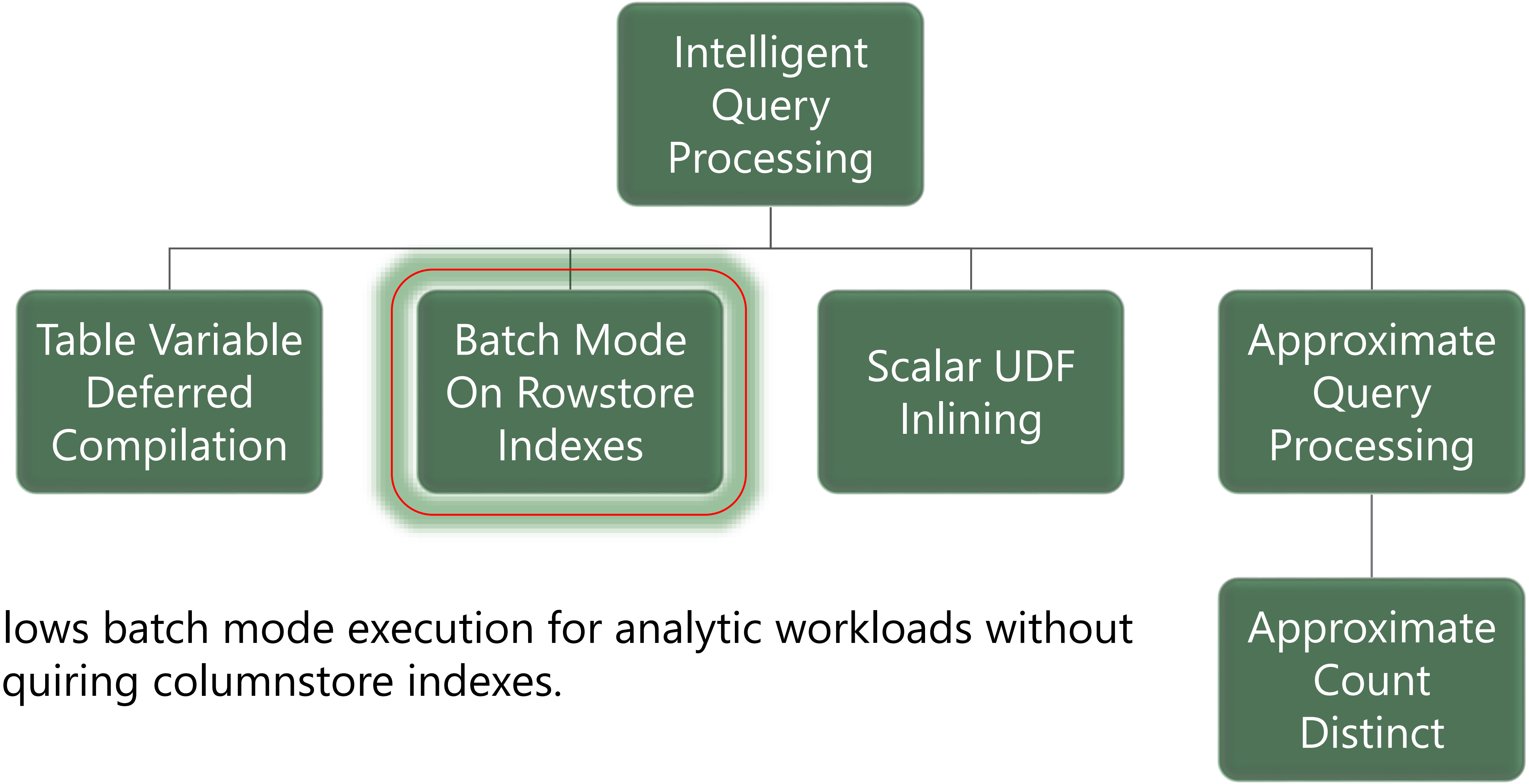


Table Variable Deferred Compilation



Queries that contain table variables will defer the compilation of the execution plan until the first execution of the query statement. This allows the execution plan to use the actual cardinality of the table variable instead of a one row guess.

Batch Mode on Rowstore Indexes



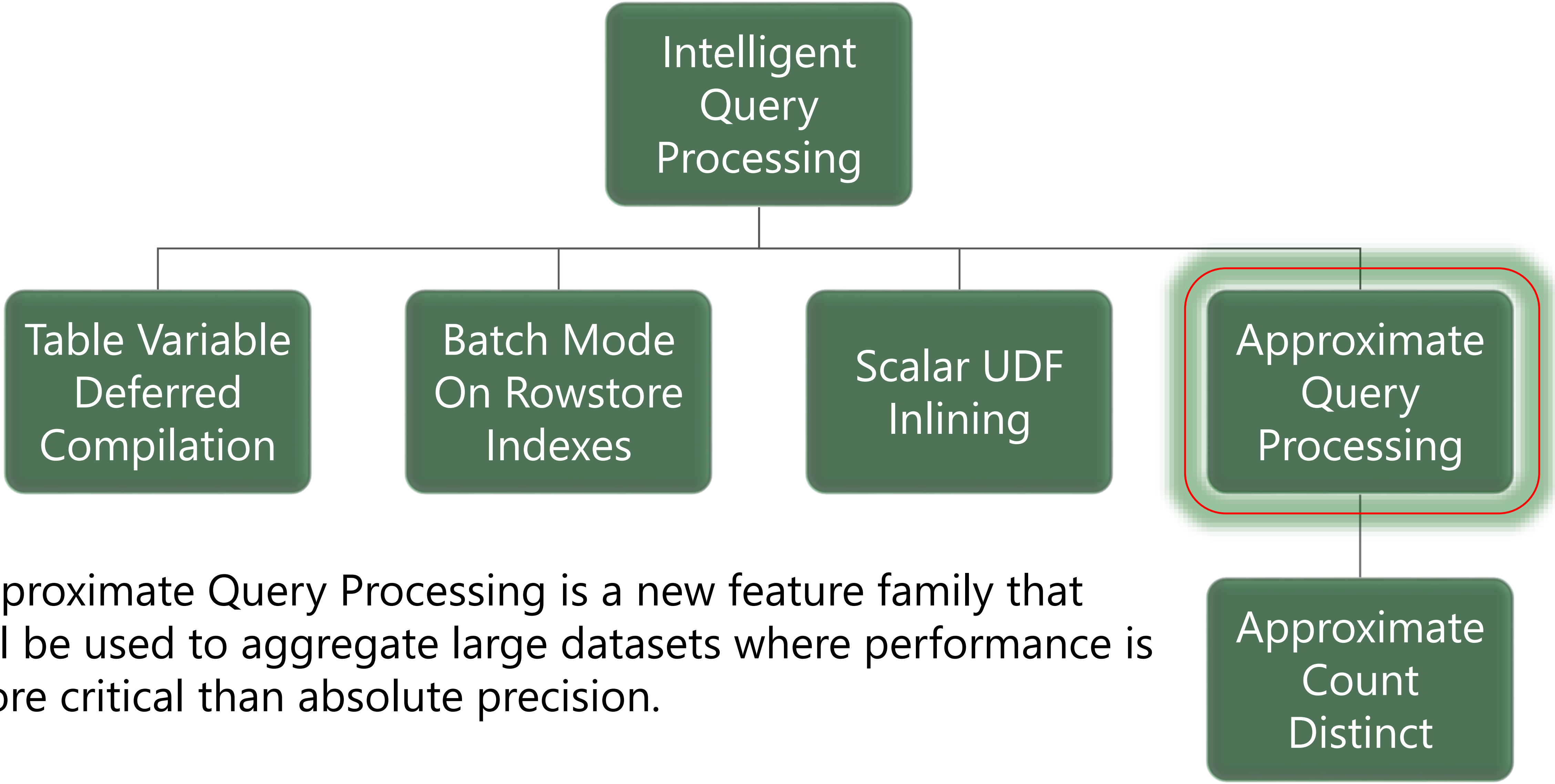
Allows batch mode execution for analytic workloads without requiring columnstore indexes.

Scalar User-Defined Function Inlining

- Intelligent Query Processing
  - Table Variable Deferred Compilation
  - Batch Mode On Rowstore Indexes
  - Scalar UDF Inlining
  - Approximate Query Processing
    - Approximate Count Distinct

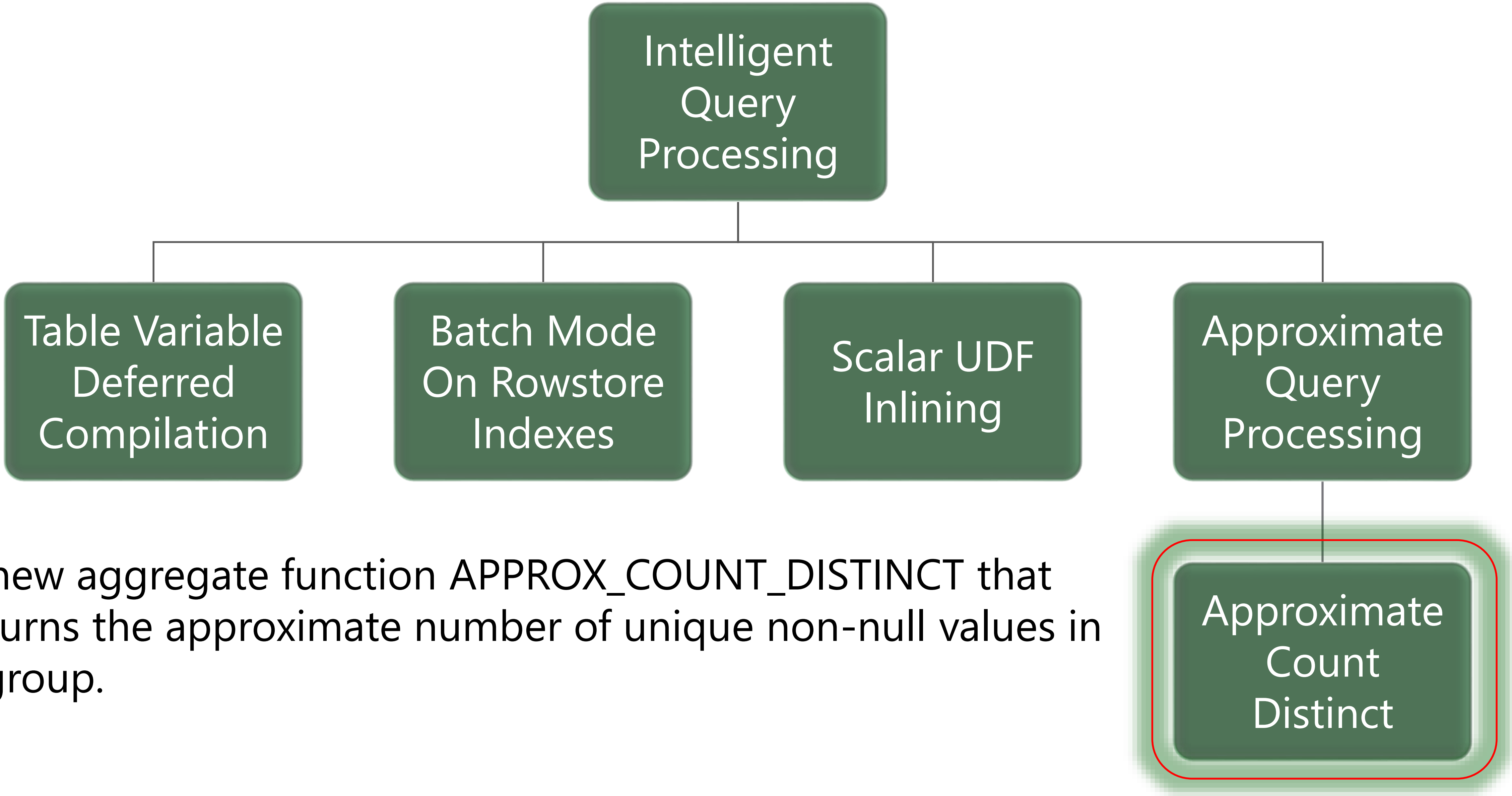
Scalar UDFs are automatically transformed into scalar expressions or scalar subqueries that are substituted in the calling query in place of the UDF operator.

Approximate Query Processing



Approximate Query Processing is a new feature family that will be used to aggregate large datasets where performance is more critical than absolute precision.

Approximate Count Distinct (2019)



A new aggregate function APPROX_COUNT_DISTINCT that returns the approximate number of unique non-null values in a group.

Approximate Count Distinct

It returns the approximate number of unique non-null values in a group.

It is designed to provide aggregations across large data sets where responsiveness is more critical than absolute precision.

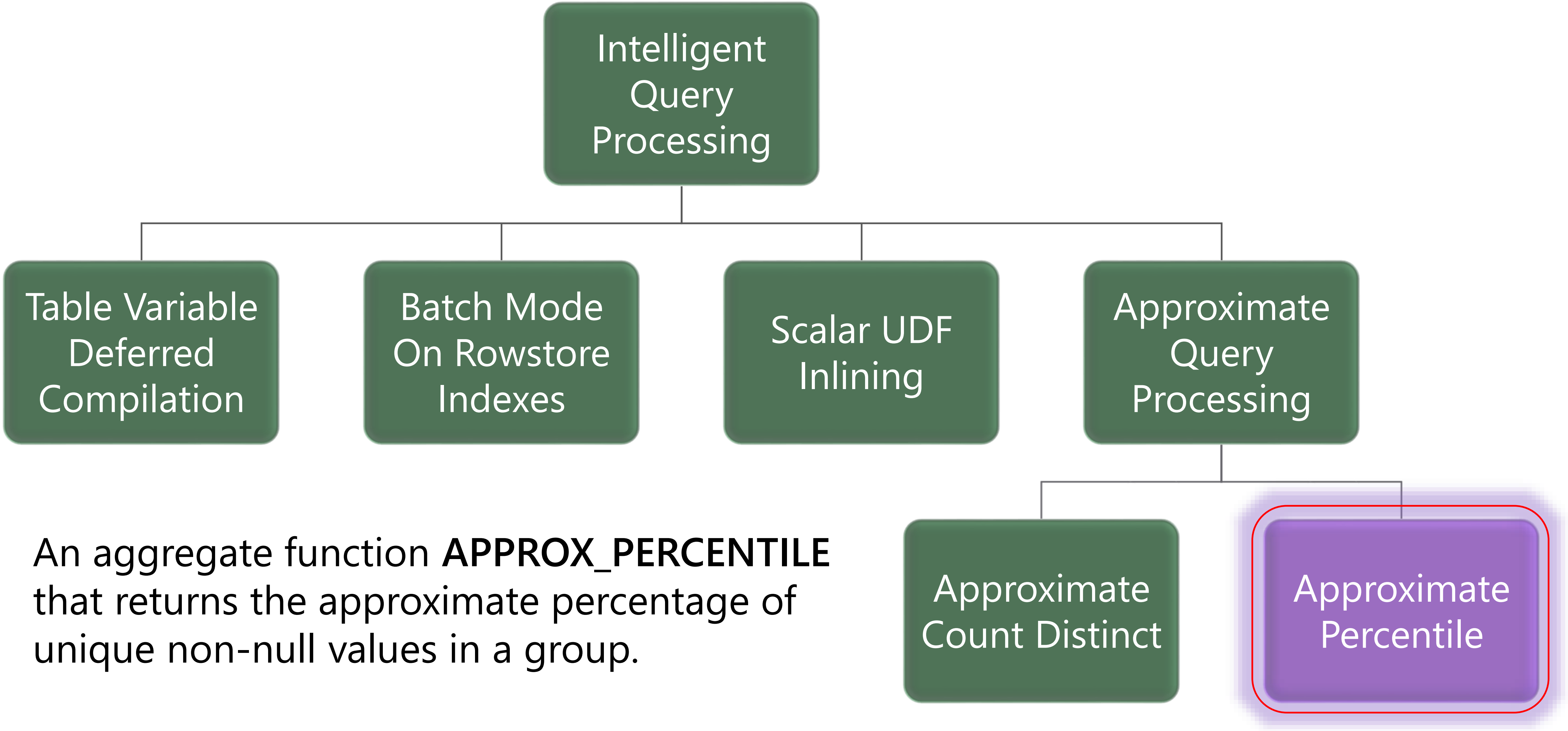
Guarantees up to a 2% error rate within a 97% probability.

Requires less memory than an exhaustive COUNT DISTINCT operation so it is less likely to spill memory to disk compared to COUNT DISTINCT.

Approximate Count Distinct

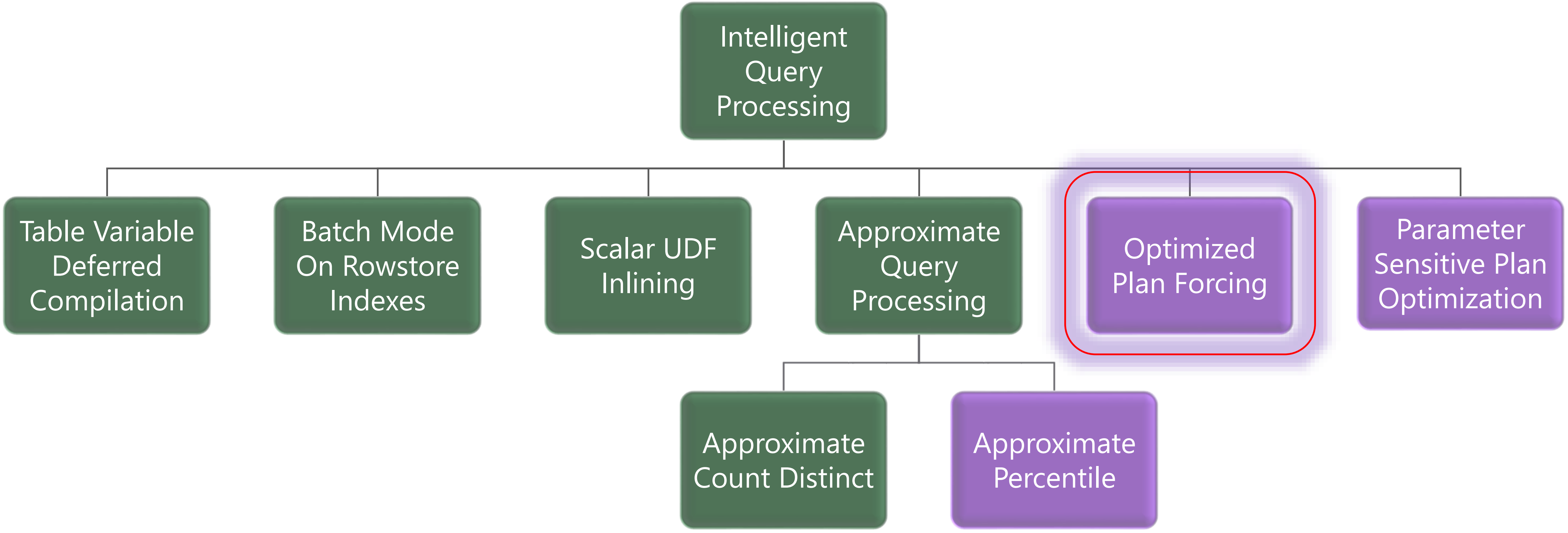
```sql
SELECT APPROX_COUNT_DISTINCT(O_OrderKey) AS Approx_Distinct_OrderKey
FROM dbo.Orders;
```

Approximate Percentile (2022)



An aggregate function **APPROX_PERCENTILE** that returns the approximate percentage of unique non-null values in a group.

Optimized Plan Forcing (2022)



Optimized Plan Forcing (2022)

Query Compilation Today

- Query optimization and compilation is a multi-phased process of quickly generating a “good-enough” query execution plan
- Query execution time includes compilation. Can be time and resource consuming
- To reduce compilation overhead for repeating queries, SQL caches query plans for re-use

Plans can be evicted from cache due to restarts or memory pressure

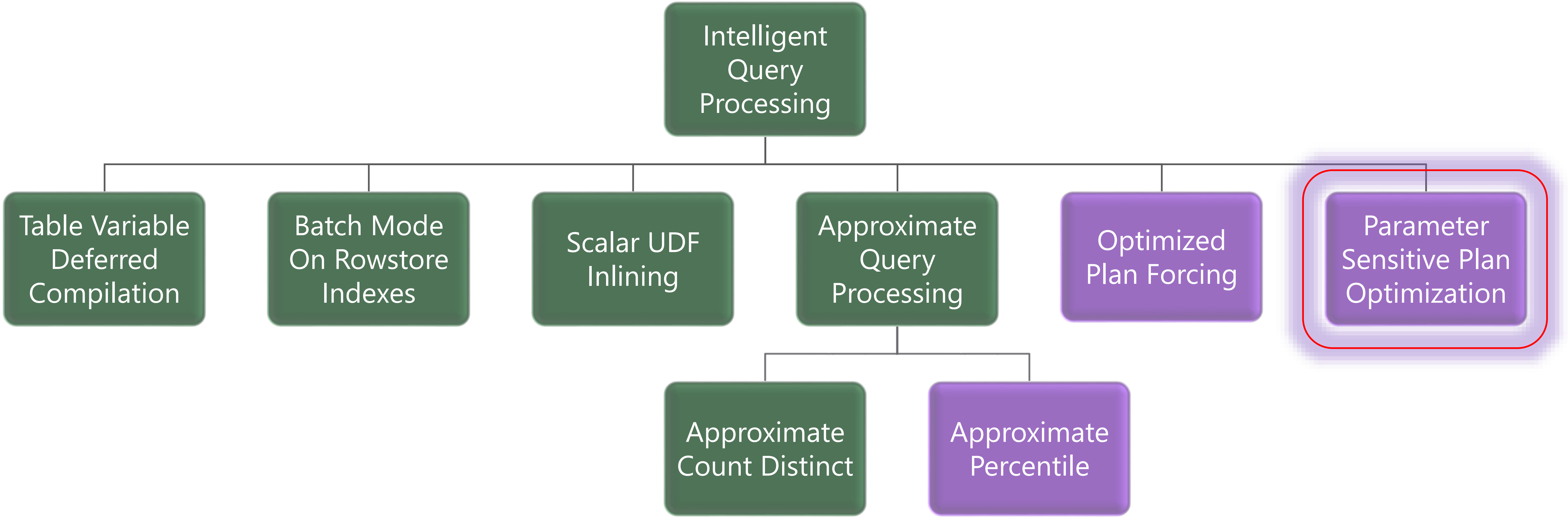
Subsequent calls to the query require a full new compilation

Optimized Plan Forcing (2022)

Query Compilation Replay

- Stores a *compilation replay script* (CRS) that persists key compilation steps in Query Store (not user visible)
- Version 1 targets previously forced plans through Query Store and Automatic Plan Correction
- Uses those previously-recorded CRS to quickly reproduce and cache the original forced plan **at a fraction of the original compilation cost**
- Compatible with Query Store hints and secondary replica support

Parameter Sensitive Plan Optimization (2022)



Parameter Sensitive Plans (2022)

Parameter Sensitive Plans Today

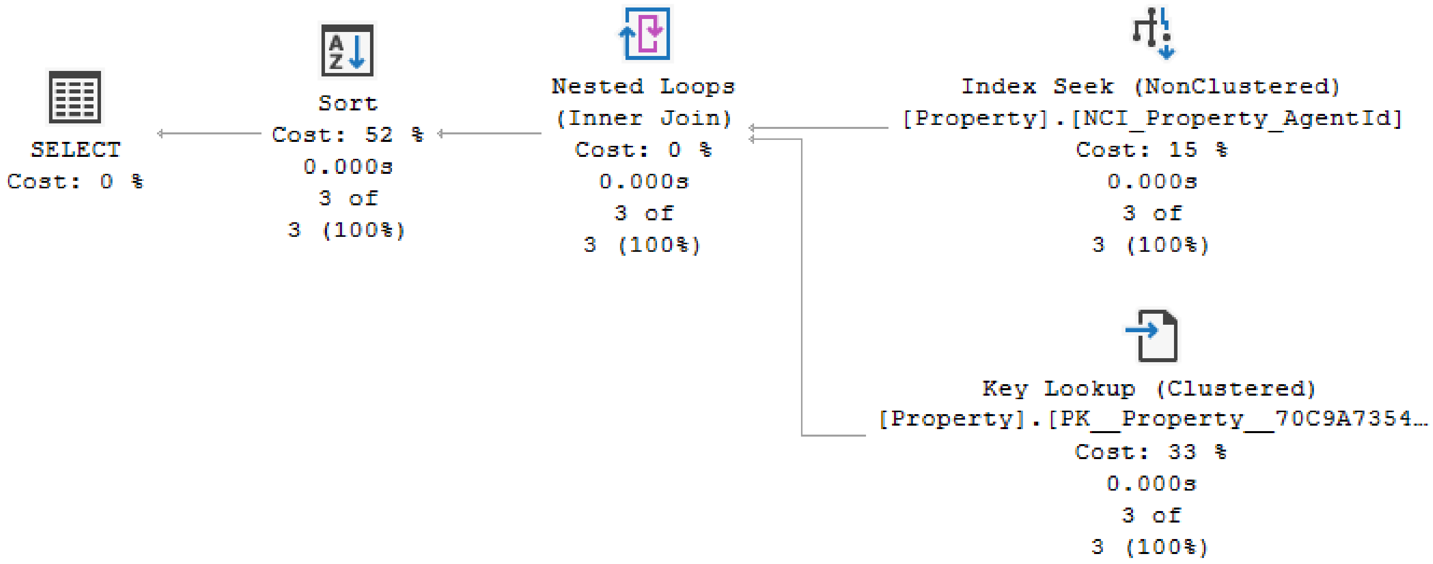
- Parameter-sniffing problem refers to a scenario where a **single** cached plan for a parameterized query is **not optimal for all** possible input parameter values
- If plan is not representative of most executions, you have a perceived "bad plan"

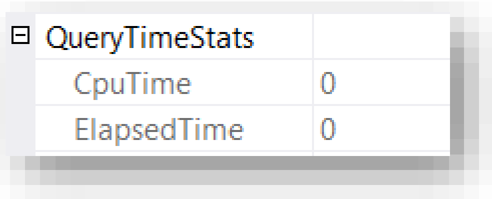
Current Workarounds

- RECOMPILE
- OPTION (OPTIMIZE FOR…)
- OPTION (OPTIMIZE FOR UNKNOWN)
- Disable parameter sniffing entirely
- KEEPFIXEDPLAN
- Force a known plan
- Nested procedures
- Dynamic string execution

PSP today (Example of Real Estate agent's portfolio)

New compile on Agent 4

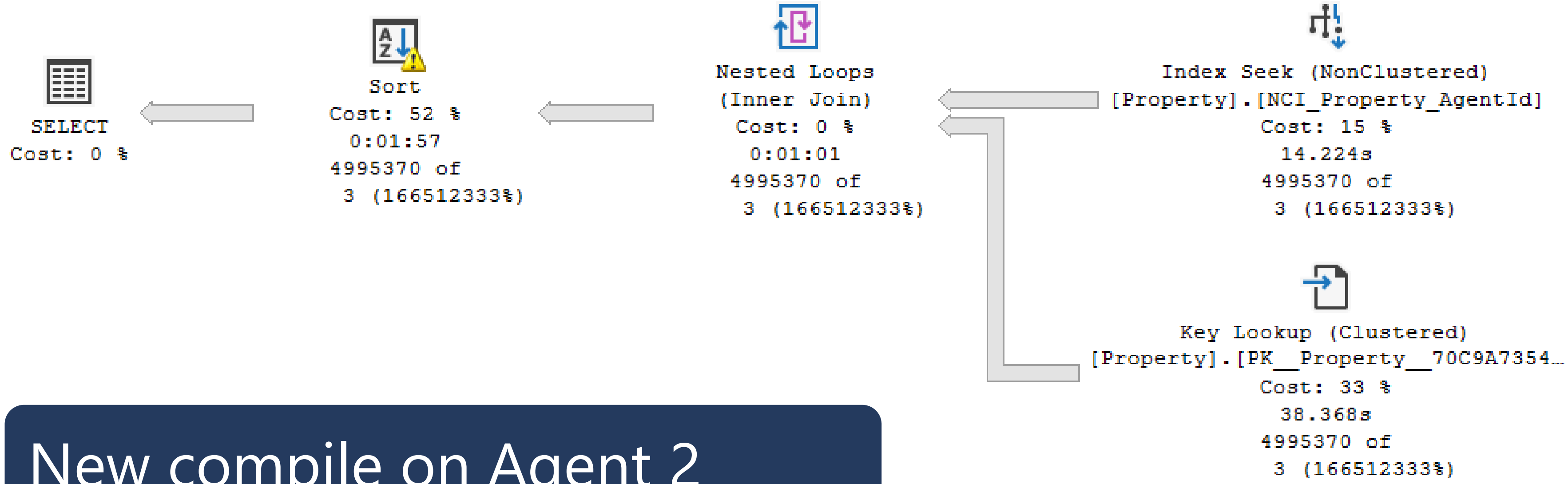




This example was borrowed from Pedro Lopes @SQLPedro

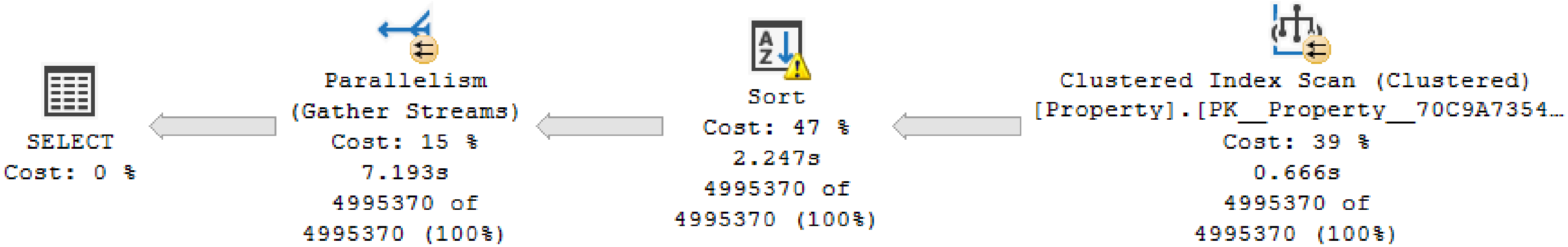
PSP today (Example of Real Estate agent's portfolio)

Using cached plan for Agent 2



QueryTimeStats	
CpuTime	88667
ElapsedTime	214222

New compile on Agent 2



QueryTimeStats	
CpuTime	46620
ElapsedTime	105288

PSP Optimization (2022)

Automatically enables multiple, active cached plans for a single parameterized statement

Cached execution plans will accommodate different data sizes based on the customer-provided runtime parameter value(s)

Design considerations

- Too many plans generated could create cache bloat, so limit # of plans in cache
- Overhead of PSP optimization must not outweigh downstream benefit
- Compatible with Query Store plan forcing

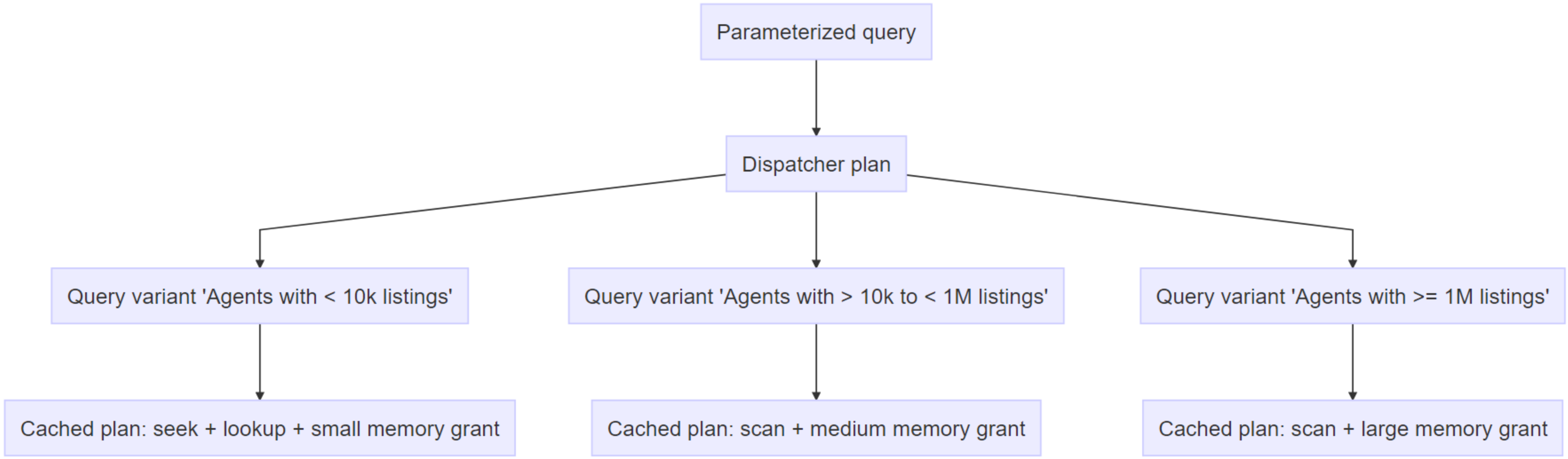
PSP Predicate Selection (2022)

During initial compilation PSP optimization will evaluate the most "at risk" parameterized predicates (up to three out of all available)

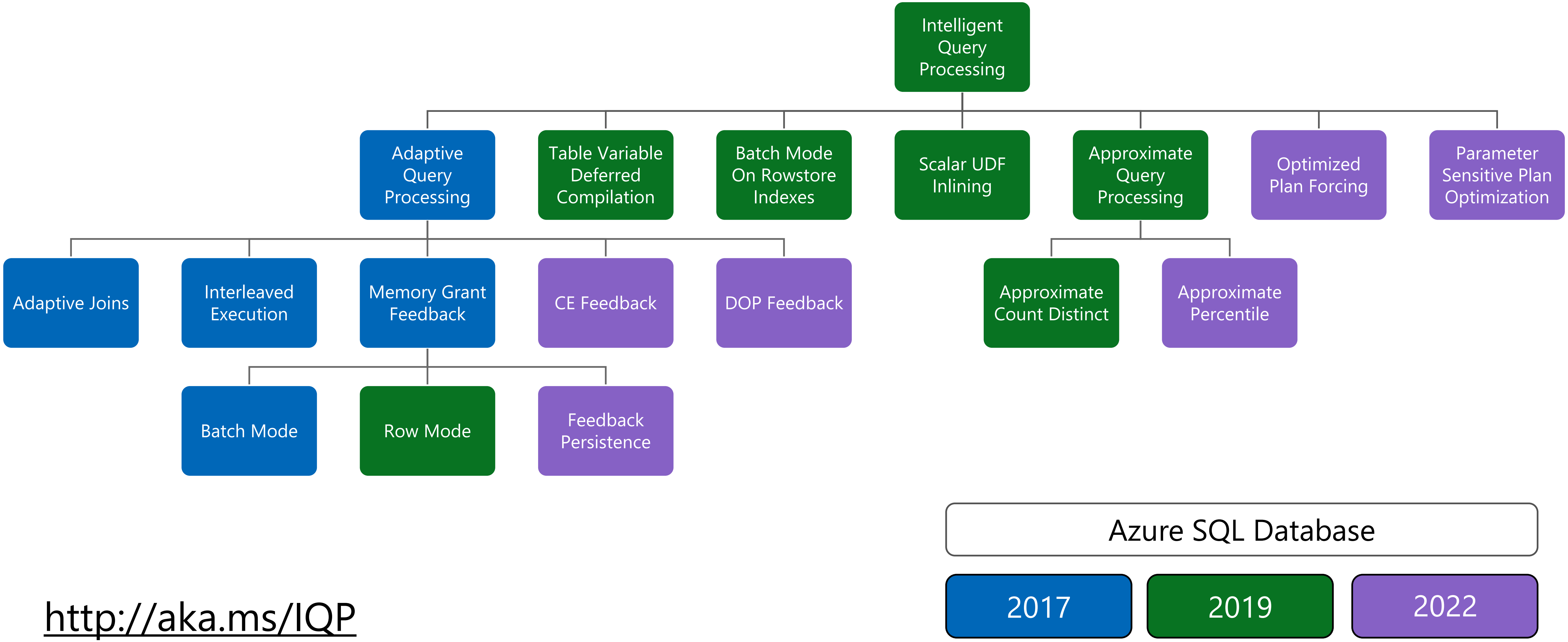
First version is scoped to equality predicates referencing statistics-covered columns; `WHERE AgentId = @AgentId`

Uses the statistics histogram to identify non-uniform distributions

Boundary Value Selection (Dispatcher Plan)



Intelligent Query Processing (2022)



http://aka.ms/IQP

Demonstration

Intelligent query processing

- Interleaved Execution
- Batch Mode on RowStore
- Memory Grant Feedback (Row Mode)

Intelligent query processing

- Observing Batch-Mode Memory Grant Feedback
- Using APPROX_COUNT_DISTINCT to improve performance
- Observing Table Variable Deferred Compilation
- Observing Scalar UDF Inlining

Questions?

Knowledge Check

Is it possible to disable Intelligent Query Processing features?

On queries not using Interleaved execution for MSTVFs. How many rows are estimated for a MSTVFs?

Does table variable deferred compilation increase the recompilation frequency?

On queries not using table variable deferred compilation. How many rows are estimated for a table variable?

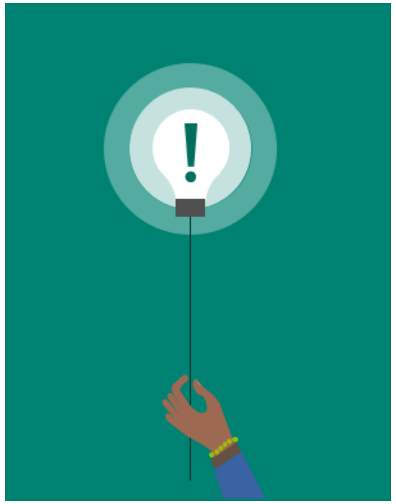
What is the minimum compatibility level that supports Batch mode on rowstore?

Extra: SQL Server Resource Governor

Objectives

After completing this learning, you will be able to:

- Understand how Resource Governor works.
- Understand how to configure Resource Governor.



Resource Governor – Feature Overview

SQL Server 2008

- Resource Governor was first introduced
- CPU and Memory Governance

SQL Server 2012

- CPU_CAP_PERCENT
- Affinity and NUMA

SQL Server 2014

- IO Governance

Feature Goals

- Minimize runaway queries
- Improve the predictable performance of queries across the SQL Server engine

Resource Governor Scenarios

Mixed Workload Types	Hosting Companies	Consolidation
A resource intensive report can take up most or all of the server resources	A customer consumes high CPU percentage, leading to complaint from other customers	Sharing resources

Benefits of Resource Governor

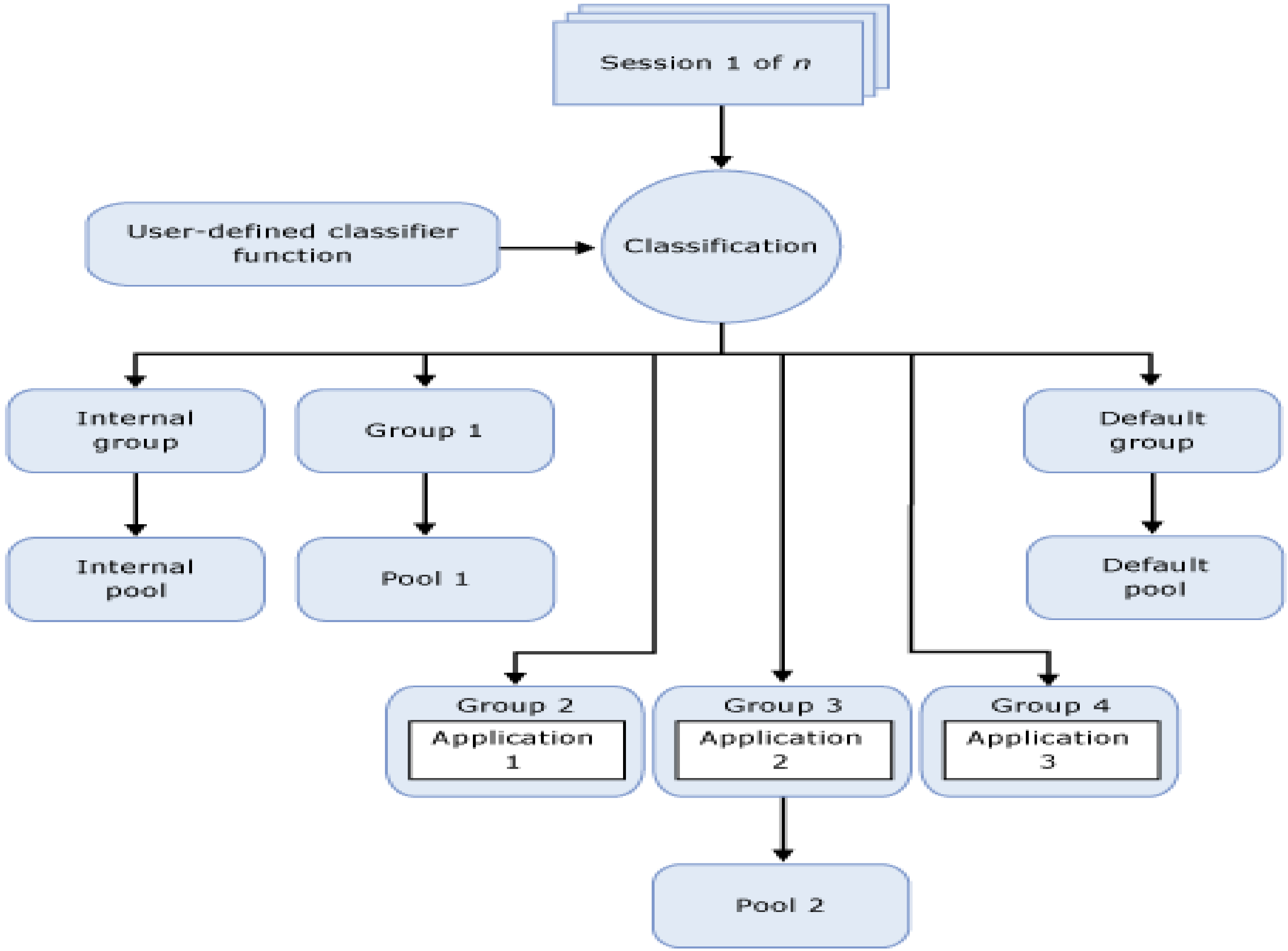
Limit large resource consumption

- Expensive reports
- Run-away queries
- Ad-hoc queries

Protect mission-critical workloads

Provide a custom monitoring solution

Resource Governor Architecture



CPU Consumption Summary

MAX can be exceeded.

Calculation of CPU is per scheduler; think of it as divided up quanta of the CPU.

Based on scheduler activity, pool settings, and if there is no CPU contention.

Importance not the same as priority.

Memory Consumption Summary

Resource Pools

- MIN_MEMORY_PERCENT
- MAX_MEMORY_PERCENT

Workload Groups

- REQUEST_MAX_MEMORY_GRANT_PERCENT
- REQUEST_MEMORY_GRANT_TIMEOUT_SEC
- MAX_MEMORY_GRANT_PERCENT

The Resource Governor can control more memory of SQL Server memory as a result of the memory redesign in SQL Server 2012.

I/O Consumption Summary

The ability to govern I/O is a critical component for SQL Server performance.

Allows full isolation of resources between workloads critical for:

- Hosting Companies
- Workload consolidation by IT

Categorization is critical for modern workloads including large scale deployments, helping to address:

- Rogue workloads
- Throwaway queries
- Maintenance operations

What I/O Can be Governed? (And What Cannot)

Can be Governed:	Cannot be Governed:
Read I/O	Internal Pool
Write I/O (some caveats)	Logical Reads
Physical Reads	Log Files
Data Files	

MAX_OUTSTANDING_IO_PER_VOLUME

```
ALTER RESOURCE GOVERNOR
{ DISABLE | RECONFIGURE } |
WITH ( CLASSIFIER_FUNCTION
     = { schema_name.function_name | NULL } ) |
RESET STATISTICS |
WITH ( MAX_OUTSTANDING_IO_PER_VOLUME = value ) [ ; ]
```

Resource Governor Monitoring Techniques

Dynamic Management Views

- sys.dm_resource_governor_resource_pools
- sys.dm_resource_governor_resource_pool_volumes

Extended Events

- File_read_enqueued
- File_write_enqueued

Performance Monitor

Resource Governor Best Practices

Enable	Enable the remote Dedicated Admin Connection (DAC)
Optimize	Optimize the Classifier function Use the PreConnect() Trace Events Test the Classifier Function Under Load Avoid Lookup Tables Avoid Expensive String Operations
Place	Place reasonable limits on the default pool and group
Use	Use memory constraints with caution

Demonstration

Resource Governor

- Configuring Resource Governor
- Examining Resource Governor Objects
- Clean-up Resource Governor

Questions?

Knowledge Check

When is the classifier function run?

What happens to the request when the classifier functions returns an invalid workload group name?

True or False: The DAC should be enabled when implementing Resource Governor.

Microsoft