Performance Features

Module 3

Learning Units covered in this Module

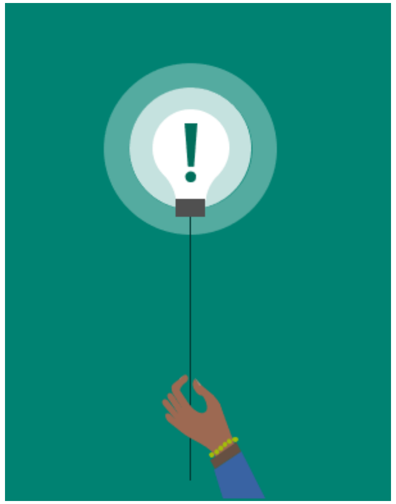
- Lesson 1: In-Memory OLTP
- Lesson 2: ColumnStore Indexes
- Lesson 3: Intelligent Query Processing
- Lesson 4: Automatic Tuning

Lesson 1: In-Memory OLTP

Objectives

After completing this learning, you will be able to:

- Explain the In memory OLTP feature in SQL Server?
- Where does Memory Optimized TempDB Metadata?

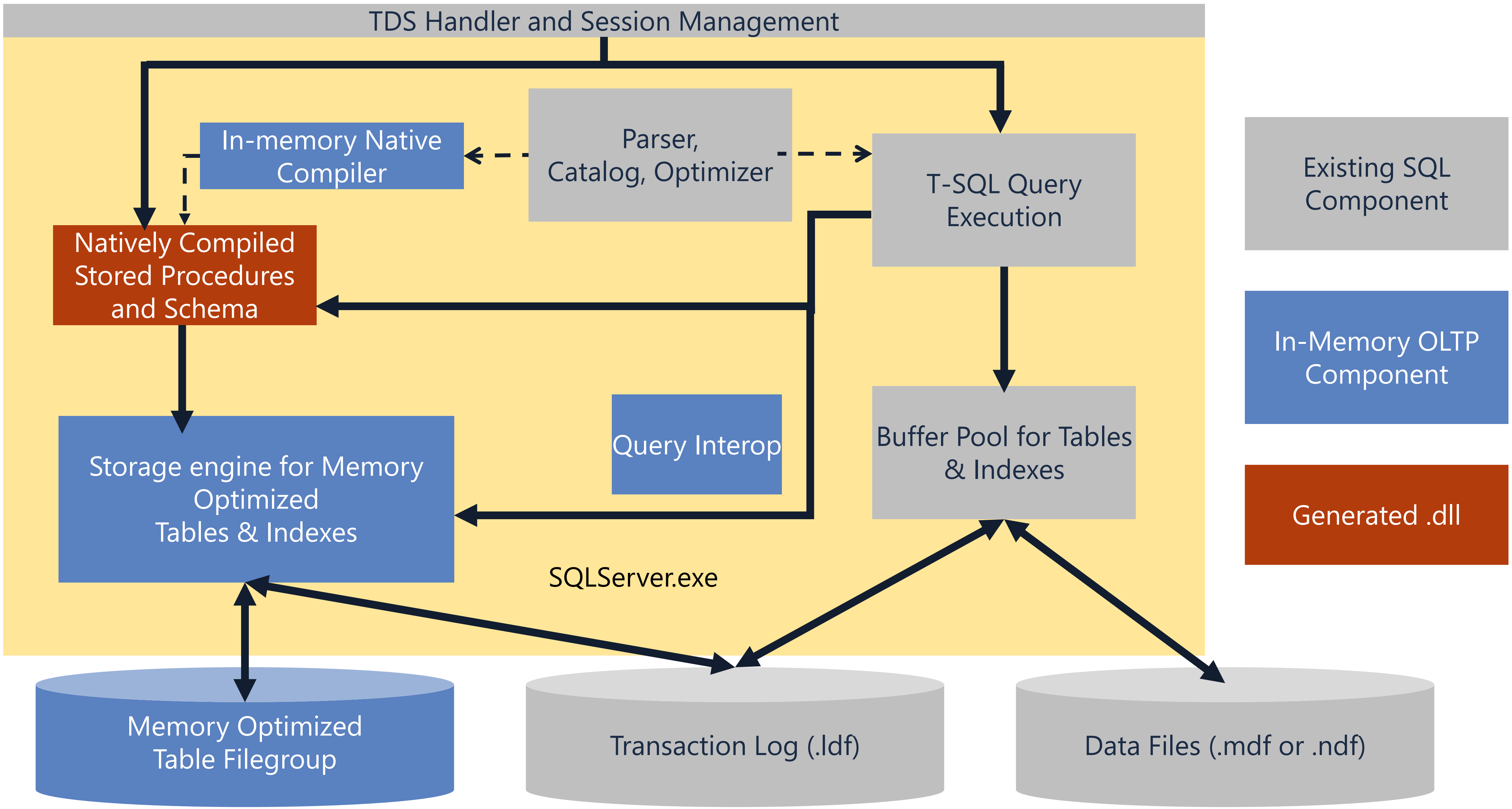


In-Memory OLTP

Architectural Pillars

Principles	Performance-critical data fits in memory	Push decisions to compilation time	Conflicts are Rare	Built-In
Architectural Pillars	**Main-Memory Optimized** • Direct pointers to rows • Indexes exist only in memory • No buffer pool • No write-ahead logging • Stream-based checkpoint and optimized logging	**T-SQL Compiled to Native Machine Code** • T-SQL compiled to machine code using VC compiler • Procedure and its queries, becomes a C function • Aggressive optimizations @ compile-time	**Non-Blocking Execution** • Multi-version optimistic concurrency control with full ACID support • Lock-free data structures • No locks, latches or spinlocks • No I/O during transaction	**SQL Server Integration** • Same manageability, administration & development experience • Integrated queries & transactions • Integrated HA and backup/restore
Results	Speed of an in-memory cache with capabilities of a database	Queries & business logic run at native-code speed	Transactions execute to completion without blocking	Hybrid engine and integrated experience

Memory-Optimized Objects



In-Memory OLTP

Memory Optimized Tables

Fully Durable by default, confirms to ACID properties of transactions.

Entire table resides in main physical memory (aka RAM).

Supports additional types for higher performance attributes

- Non-Durable – data is not persisted on disk.
- Durable with durability delayed – data is persisted but possible data loss.

Uses Row Versioning via Snapshot Isolation Level to manage concurrency.

In Memory table valued parameters (TVP) as an alternative to table variables.

In-Memory OLTP

Rows and Indexes

Rows

- The row structure is optimized for memory access.
- There are no pages.
- Rows are versioned and there are no in-place updates.

Indexes

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

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

In-Memory OLTP

Indexes

Hash Indexes

- Puts rows into buckets
- Good for Lookups
- No Ordered Scans
- Duplicates will reduce performance

Non-Clustered Indexes

- Orders data on index key
- Good for Range Scans
- Good for inequality filters
- Only does **forward** scans

In-Memory OLTP

Durability

SCHEMA_AND_DATA	• Full durability • Data changes are logged and recoverable
SCHEMA_ONLY	• Data is not persisted to disk • After a restart, the table is empty

In-Memory OLTP

Accessing Memory Optimized Tabled

Interpreted Transact-SQL	• Can be used with disk based and in-memory tables • Less optimizations for querying in-memory tables than natively compiled stored procedures
Natively compiled stored procedures	• In-memory tables only • Precompiled • Best performance

How to speed up temp tables and Table Variables by using memory optimization?

Basics of Memory-Optimized Table Variables

- Memory-optimized table variables provide efficiency using memory-optimized algorithms and data structures.
- They are stored only in memory, involve no IO activity, tempdb utilization, or contention.
- If you use temporary tables, table variables, or table-valued parameters, consider conversions of them to leverage memory-optimized tables and table variables to improve performance. The code changes are usually minimal.
- Can be passed into a stored procedure as a table-valued parameter (TVP).
- Must have at least one index, either hash or nonclustered.

How to speed up temp tables and Table Variables by using memory optimization?

Replace Global Tempdb ##table

- Global temporary tables are replaced with memory-optimized SCHEMA_ONLY tables created at deployment time
- They can use Row Level security on SessionID Level to separate workloads from other users

Replace Session Tempdb #table

- No need for DROP TABLE #tempSessionC statements they can be replaced with DELETE FROM dbo.soSessionC or Truncate

Table Variable can be MEMORY_OPTIMIZED=ON

- Traditional table variables represent a table in the tempdb database.
- Converting to memory-optimized table variables can significantly improve performance.

Convert Inline to Explicit

- Inline syntax for table variables does not support memory-optimization
- Converting inline syntax to explicit syntax for the TYPE definition is recommended for memory optimization

Memory Optimized TempDB Metadata

Problem: High multi-user rates of tempdb usage can lead to latency due to…

- **GAM/SGAM Allocation Contention –** Multiple users needing to allocate pages for temp tables
- **System table page latch waits –** High rates of create/drop require system table modifications

Solution: Memory Optimized TempDB Metadata

- Key tempdb system tables become SCHEMA_ONLY memory optimized tables
- Latch and lock free
- Turn on with ALTER SERVER CONFIGURATION
- This is NOT user data, just metadata so memory requirements are small

```
ALTER SERVER CONFIGURATION SET MEMORY_OPTIMIZED TEMPDB_METADATA = ON;
```

Questions?

Knowledge Check

What are the two ways to access data in memory optimized tables?

What are the two types of indexes that can be created on Memory optimized Tables?

Which Durability mode will reduce overhead of logging transactions and writing data to disk?

What In Memory feature can be used as an alternative to table variables?

How does In Memory OLTP feature reduce concurrency bottlenecks?

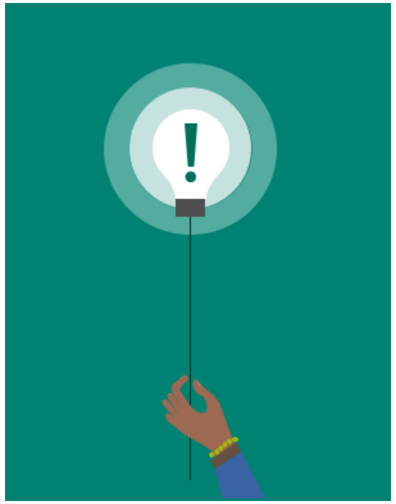
True/False? In Memory OLTP feature does not support ACID transaction attributes.

Lesson 2: ColumnStore Indexes

Objectives

After completing this learning, you will be able to:

- Columnstore Indexes
- Types of Columnstore Indexes
- Columnstore index architecture



Row Store & Column Store

ROW STORE

COLUMN STORE

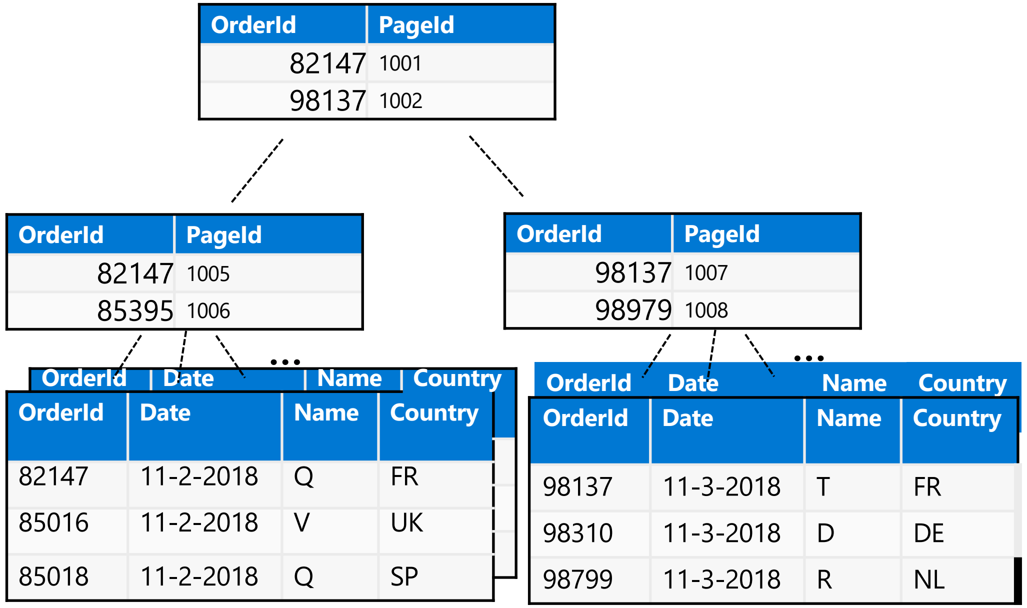
Rowstore vs Columnstore Tables

Logical table structure

OrderId	Date	Name	Country
85016	11-2-2018	V	UK
85018	11-2-2018	Q	SP
85216	11-2-2018	Q	DE
85395	11-2-2018	V	NL
82147	11-2-2018	Q	FR
86881	11-2-2018	D	UK
93080	11-3-2018	R	UK
94156	11-3-2018	S	FR
96250	11-3-2018	Q	NL
98799	11-3-2018	R	NL
98015	11-3-2018	T	UK
98310	11-3-2018	D	DE
98979	11-3-2018	Z	DE
98137	11-3-2018	T	FR
…	…	…	…

Clustered/Non-clustered rowstore index

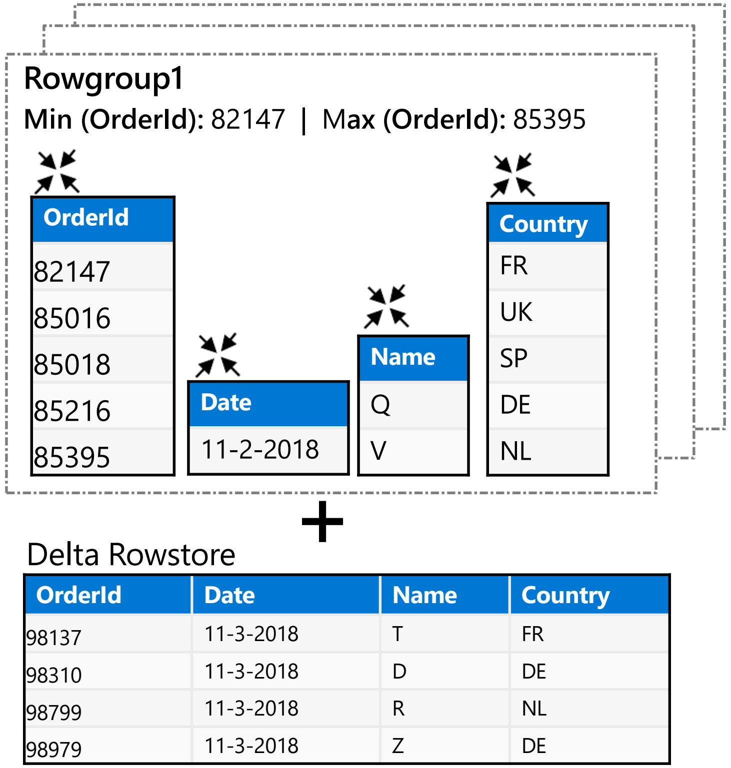
(OrderId)



- Data is stored in a B-tree index structure for performant lookup queries for particular rows.
- Clustered rowstore index: The leaf nodes in the structure store the data values in a row (as pictured above)
- Non-clustered (secondary) rowstore index: The leaf nodes store pointers to the data values, not the values themselves

Clustered columnstore index

(OrderId)



- Data stored in compressed columnstore segments after being sliced into groups of rows (rowgroups/micro-partitions) for maximum compression
- Rows are stored in the delta rowstore until the number of rows is large enough to be compressed into a columnstore

Columnstore Indexes

What are Columnstore Indexes

Columnstore indexes

- Good for OLAP workloads
- Benefits heavy workloads that perform many table and index scans

In-Memory OLTP can be combined with Columnstore technology

Intended to speed up queries that read large amounts of data

Great for data warehousing scenarios

- Write once, read many times

Data arranged by column

- Traditional data pages are arranged by row
- Compression opportunities when data is organized column
- Column elimination

Columnstore Index Types

SQL Server 2012

- Only Non-Clustered, Non-Updatable Columnstore Indexes.
- Only available in Enterprise Edition.

SQL Server 2014

- Introduced Updatable, Clustered Columnstore Indexes
- Only available in Enterprise Edition.

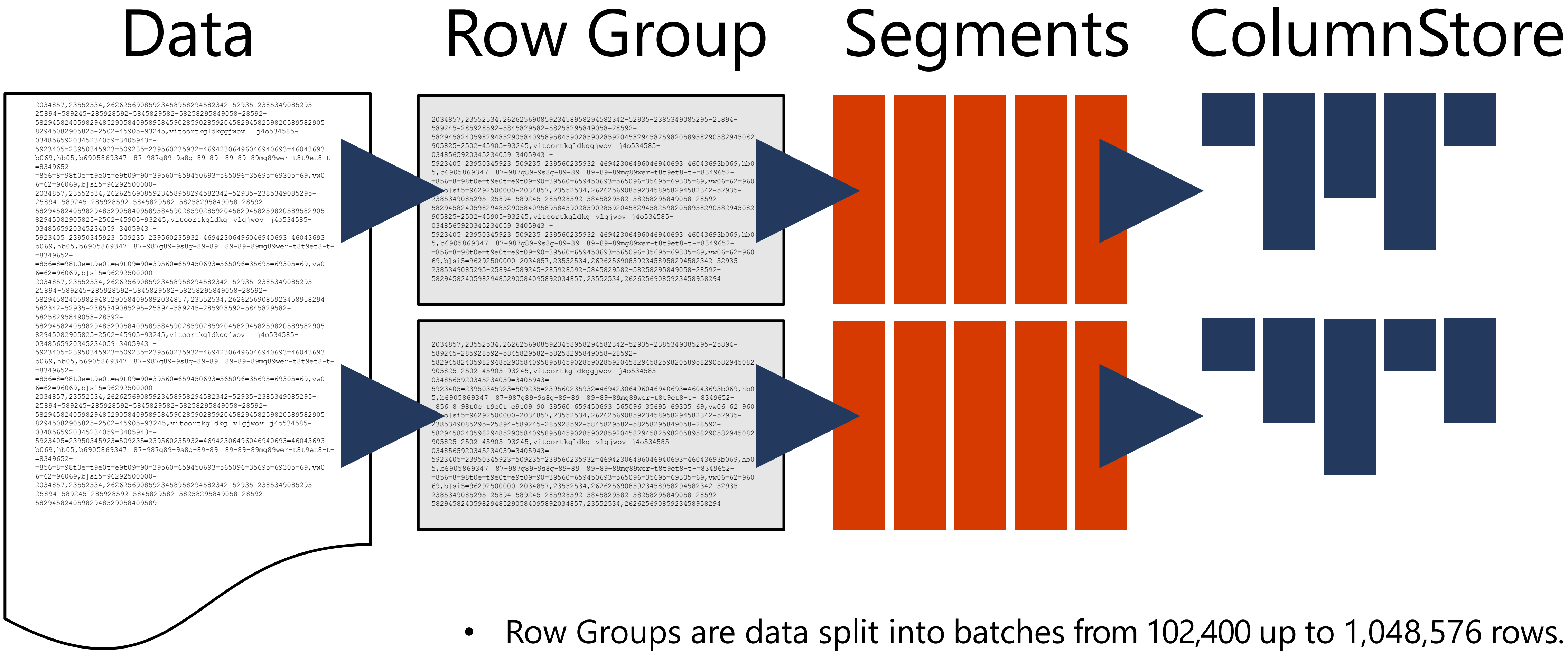
SQL Server 2016

- Introduced Updatable, Non-Clustered Columnstore Indexes
- Available on Standard Edition. (Service Pack 1)

SQL Server 2019

- Online rebuilds for Clustered Columnstore Indexes.

Columnstore Taxonomy



- Row Groups are data split into batches from 102,400 up to 1,048,576 rows.
- Segments split row groups into segments.
- Columnstore will then compress each segment.

Columnstore Indexes

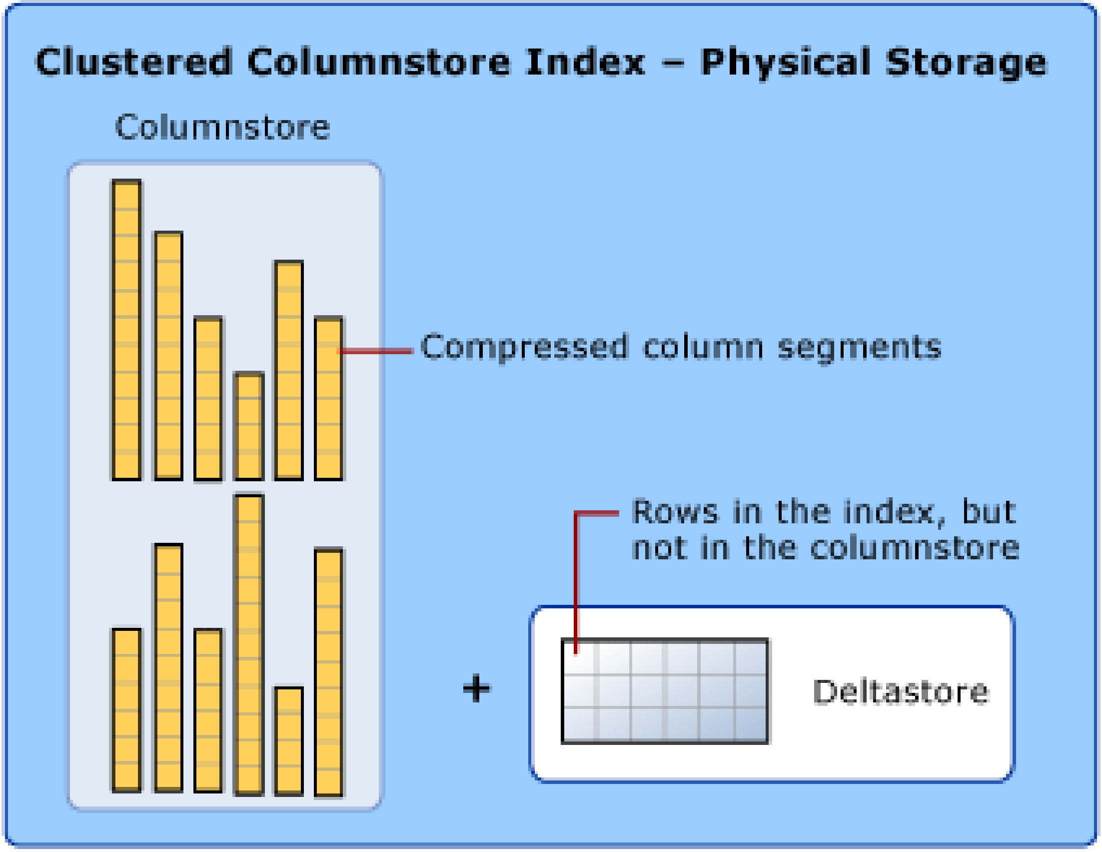
Columnstore Index Architecture

Row groups

- Rows stored in groups of up to ~1 million
- Contain a compressed column segment for each column in the index
- The delete bitmap marks rows when deleted

Delta Store

- Rowstore to temporarily house inserted records
- Will convert to row group when enough rows are inserted



Segment Elimination

Skips large chunks of data to speed up scans

Each partition in a columnstore index is broken into segments

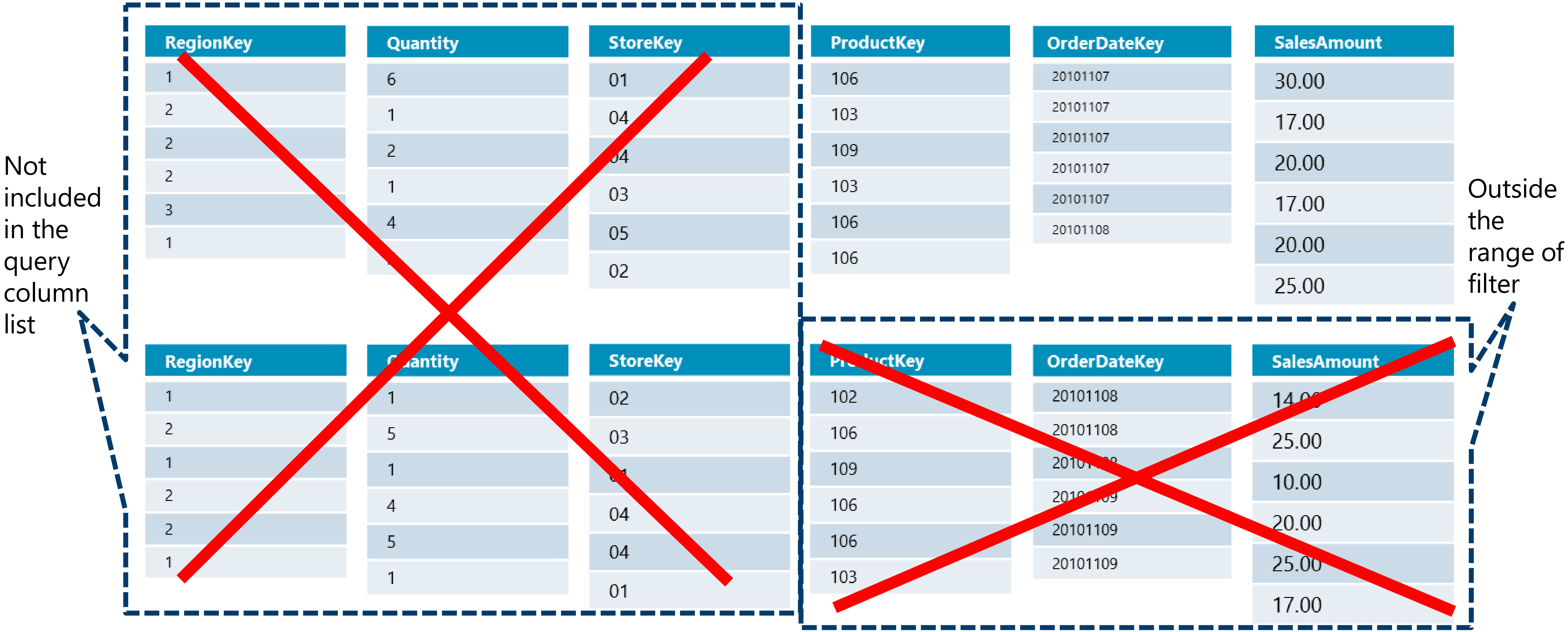
Each segment has metadata that stores the minimum and maximum value of each column for the segment

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.

Fetch Only Needed Segments

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



Demonstration

Columnstore Indexes

- Demonstrate the performance impact of Columnstore indexes on query performance

Questions?

Knowledge Check

Which version of SQL Server first introduced Updateable Clustered Columnstore Indexes?

Which version of SQL Server allows adding non clustered rowstore indexes to a clustered columnstore index?

True/False? Columnstore indexes read compressed data from disk, which means fewer bytes of data need to be read into memory?

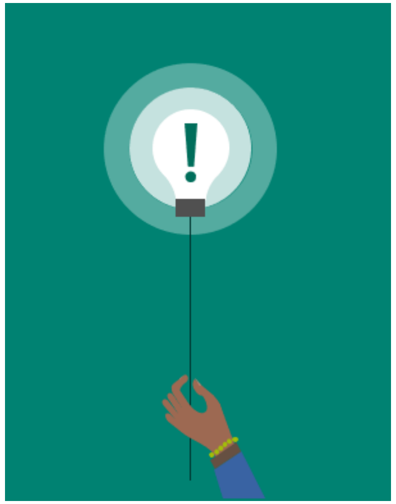
What kind of queries would benefit most from Columnstore indexes?

Lesson 3: 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.



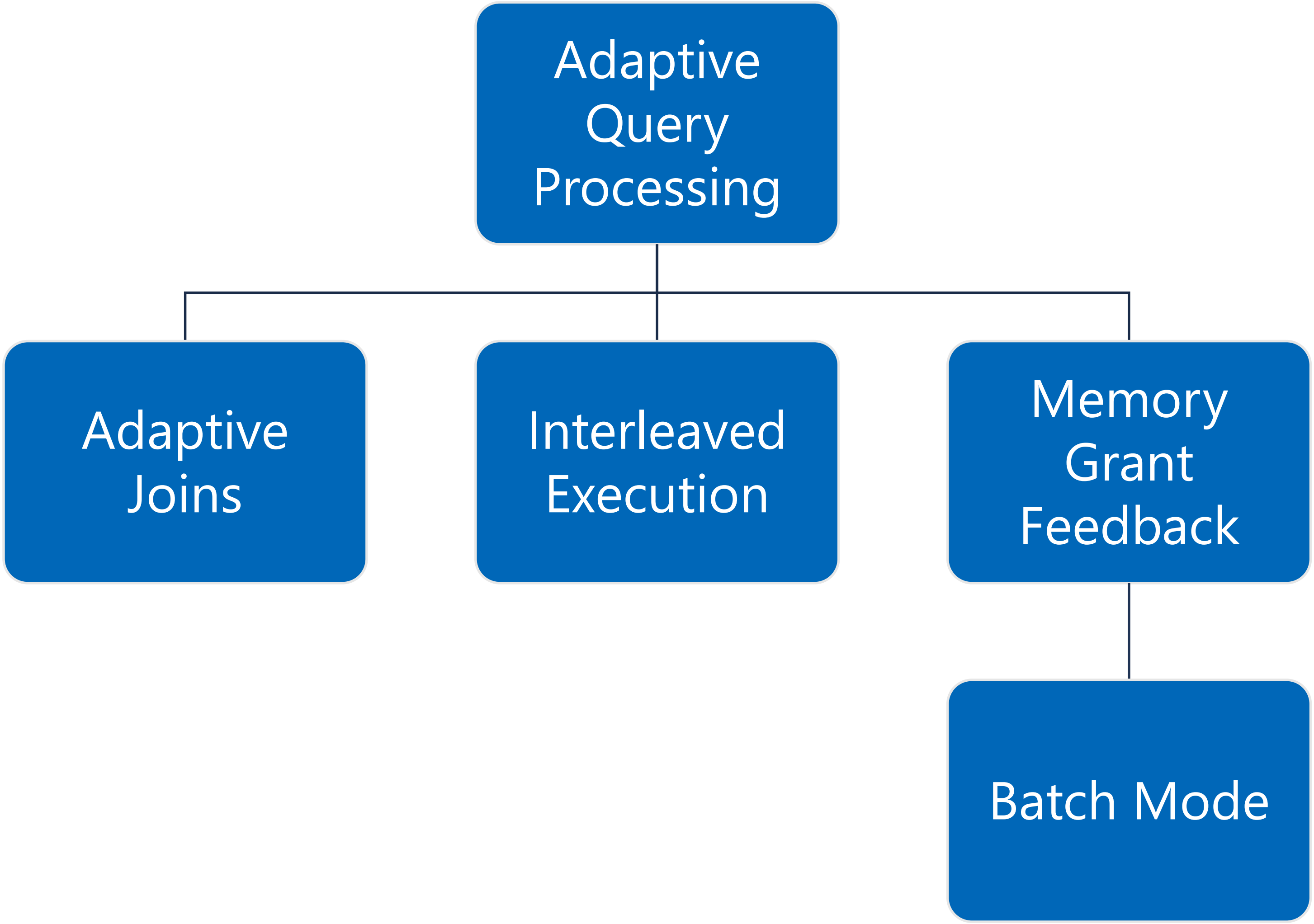
A History of Intelligent Query Processing

Adaptive Query Processing (2017)

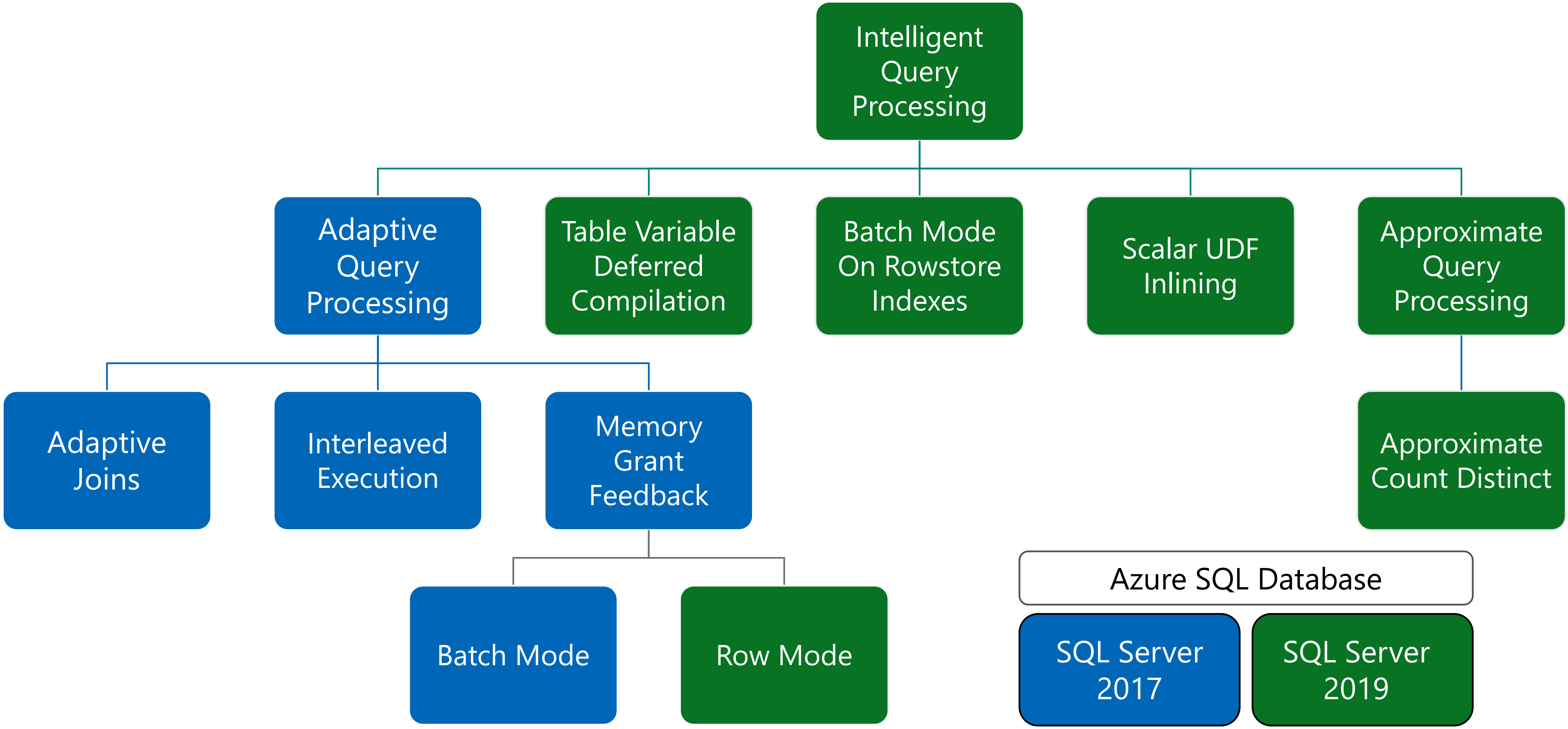
Intelligent Query Processing (2019)

New Features of IQP (2022)

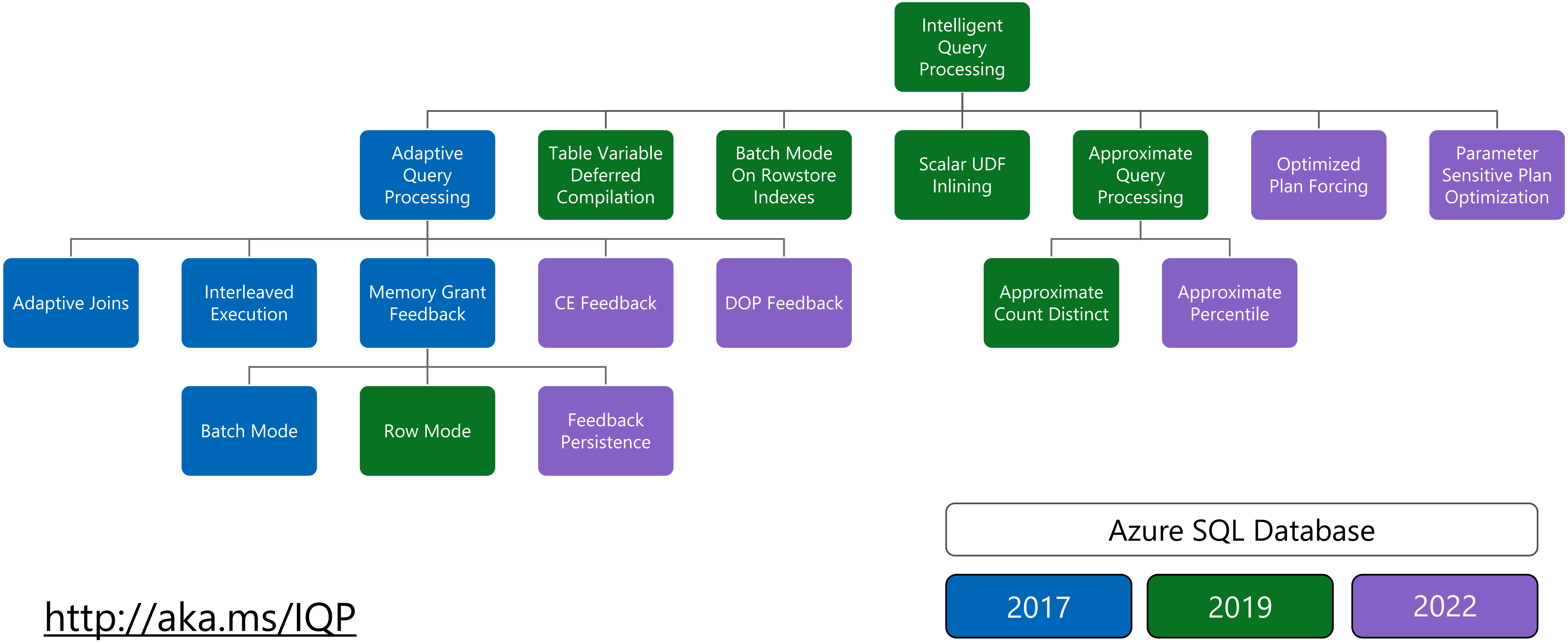
Adaptive Query Processing (2017)



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.

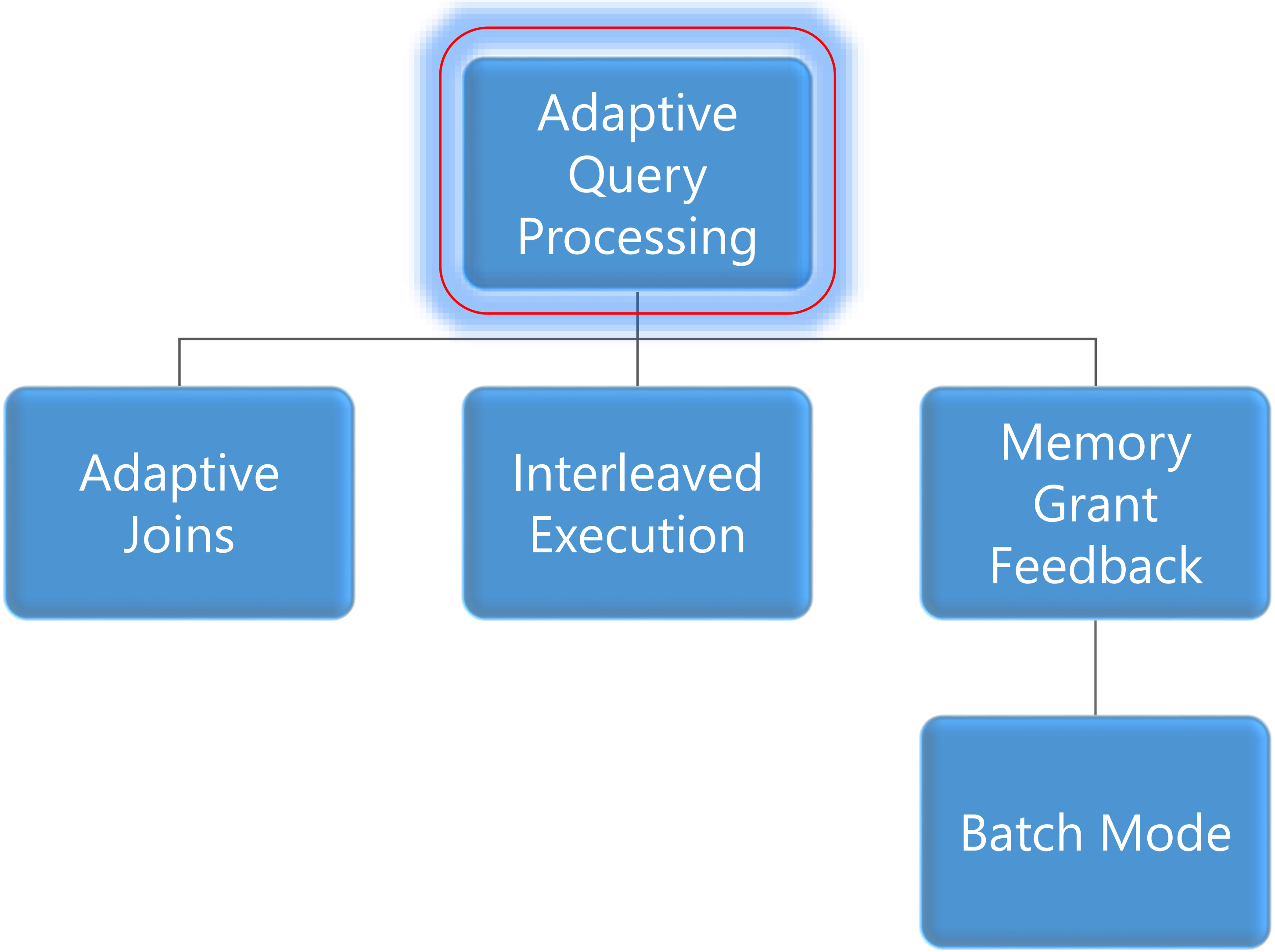
```
<statement>
OPTION (USE HINT('DISABLE_BATCH_MODE_ADAPTIVE_JOINS'));
```

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

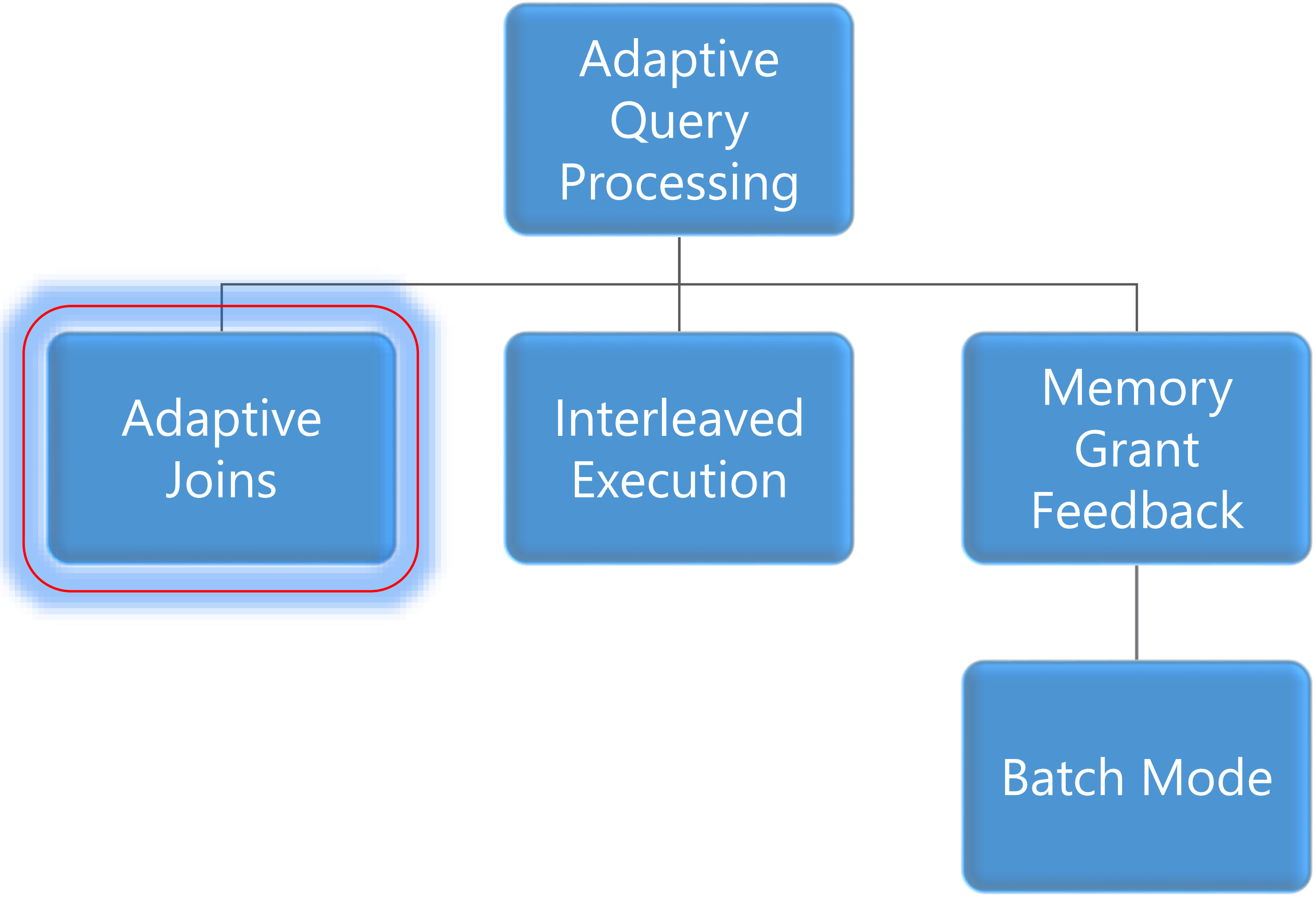
Adaptive Query Processing (2017)



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)



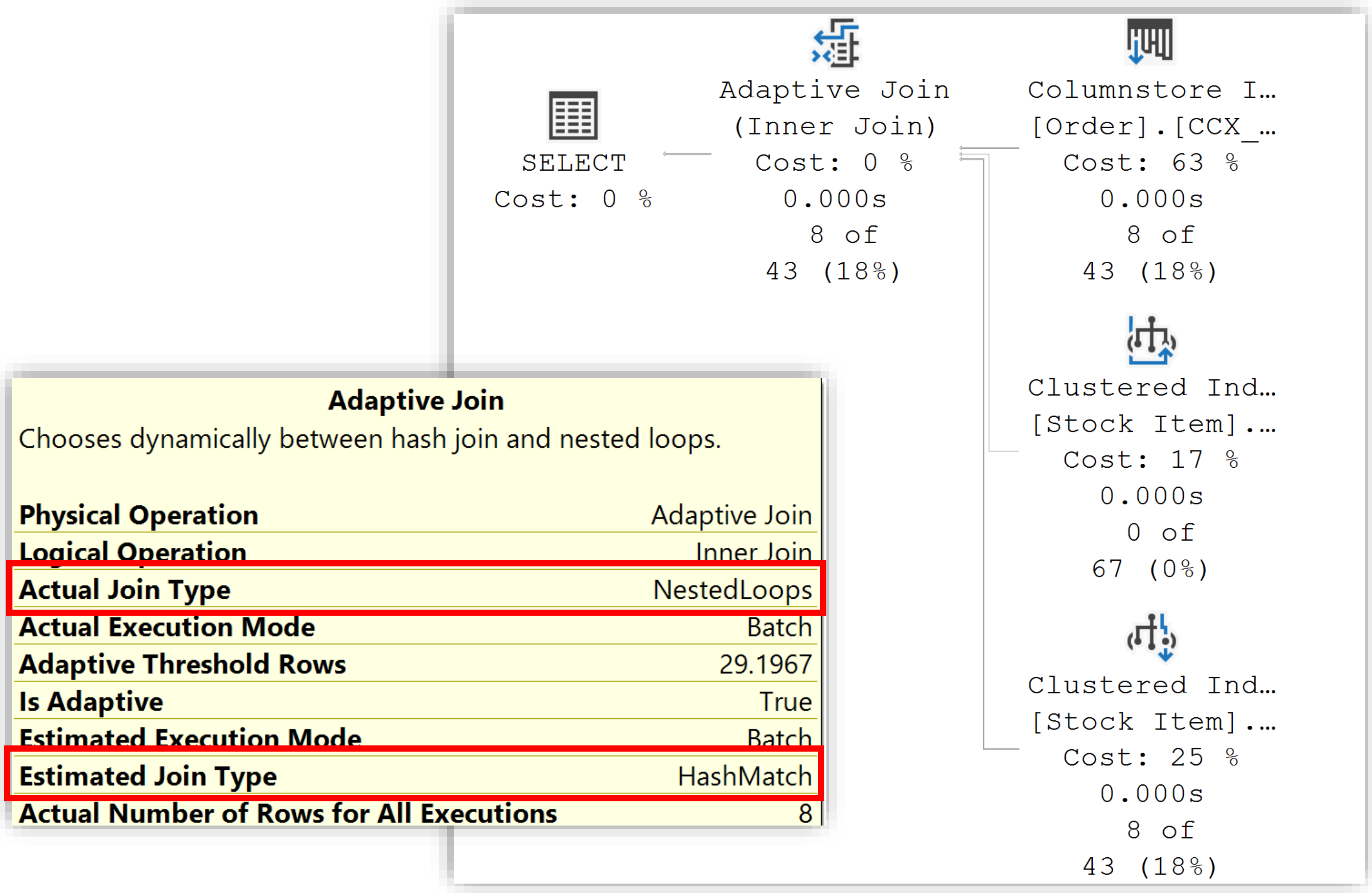
This feature enables the choice of either the Hash or the Nested Loop join type.

Decision is deferred until statement execution.

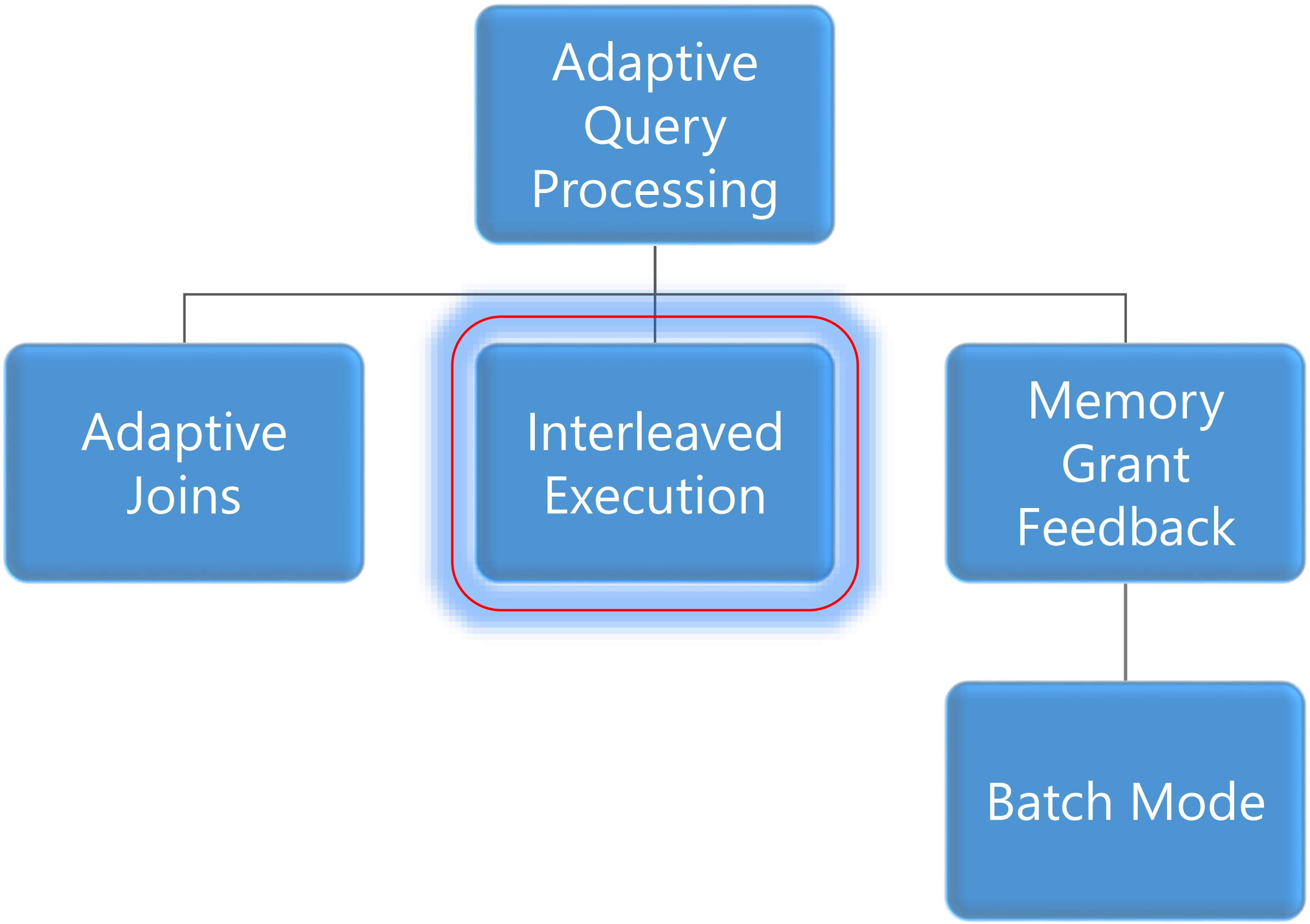
No need to use join hints in queries.

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.

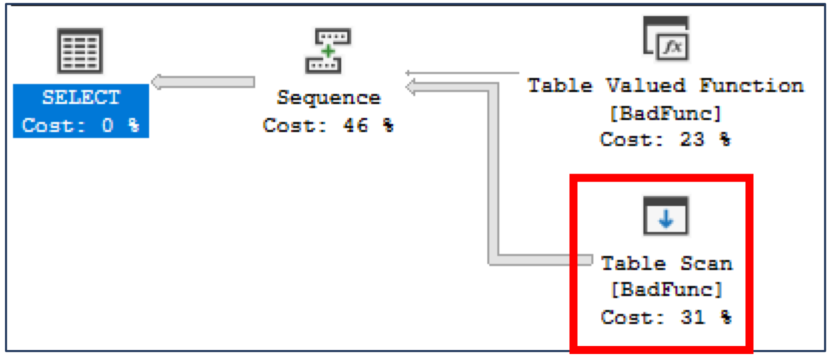
Now execution is paused so a better cardinality estimate can be captured.

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

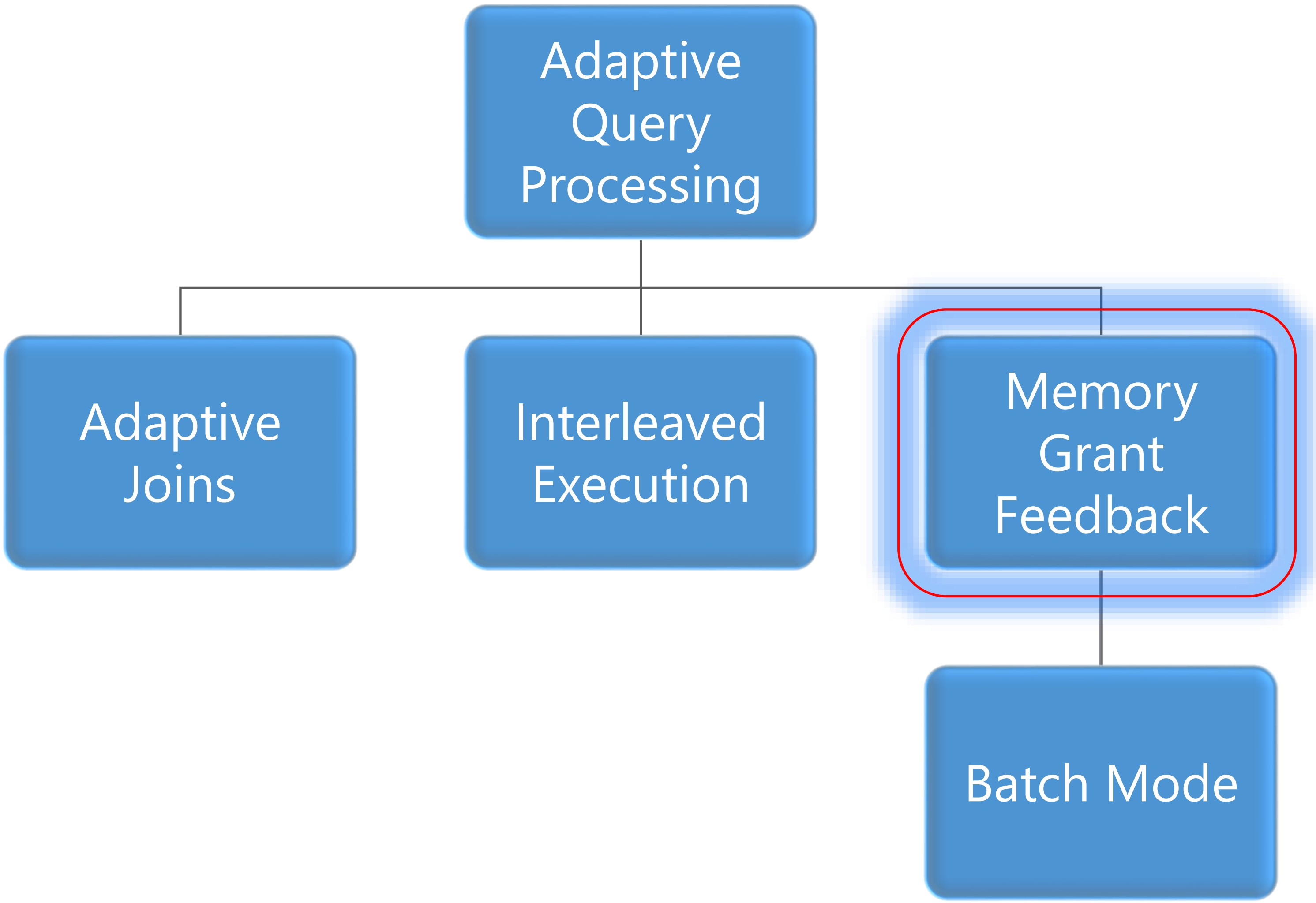


During optimization if SQL Server encounter a read-only multi-statement table-valued function (MSTVF), it will pause optimization, execute the applicable subtree, capture accurate cardinality estimates, and then resume optimization for downstream operations.

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)



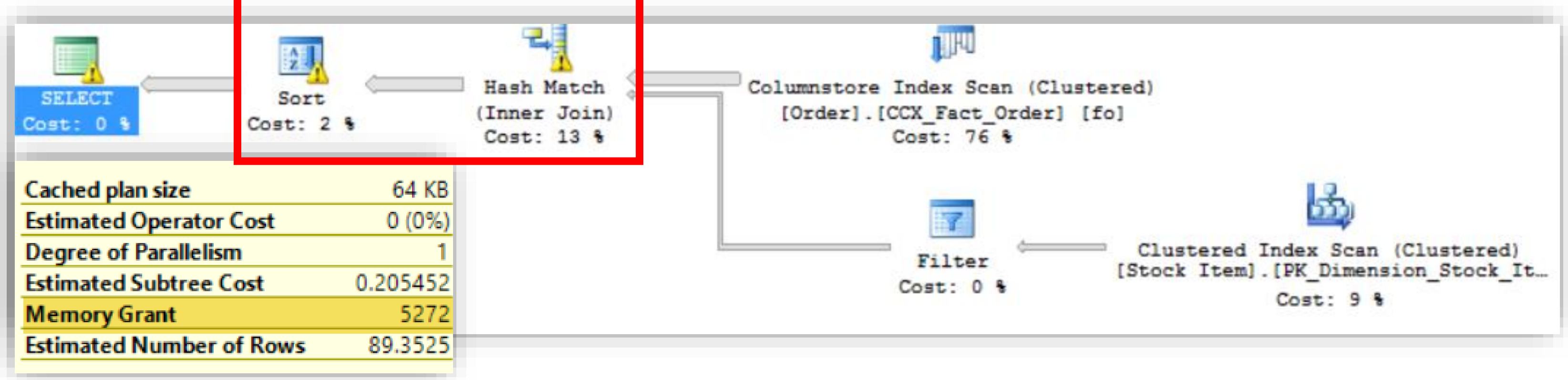
When compiling an execution plan, the query engine estimates how much memory is needed to store rows during join and sort operations.

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

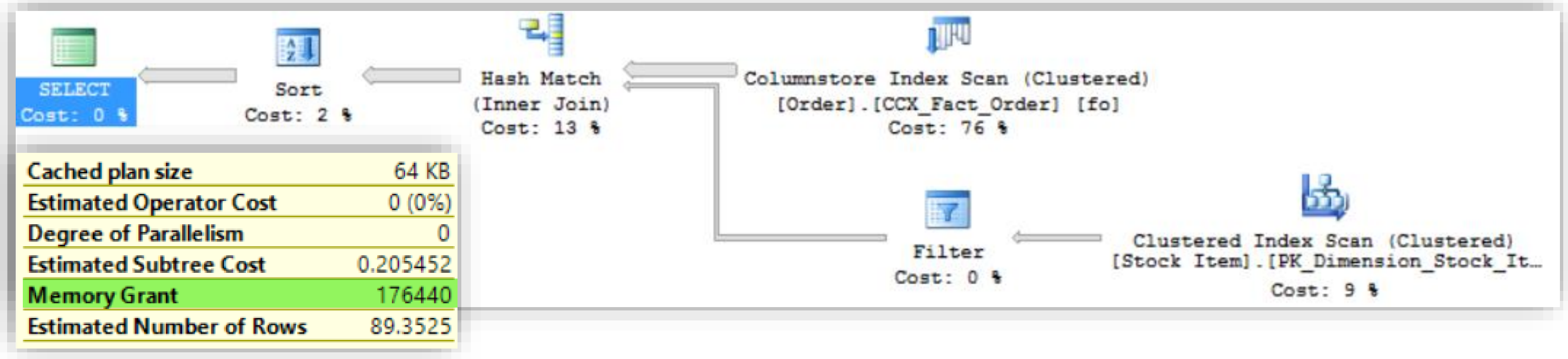
This feature recalculates memory on first execution and updates the cached plan.

Batch Mode Memory Grant Feedback (2017)

First Execution (Spills detected; feedback generated)

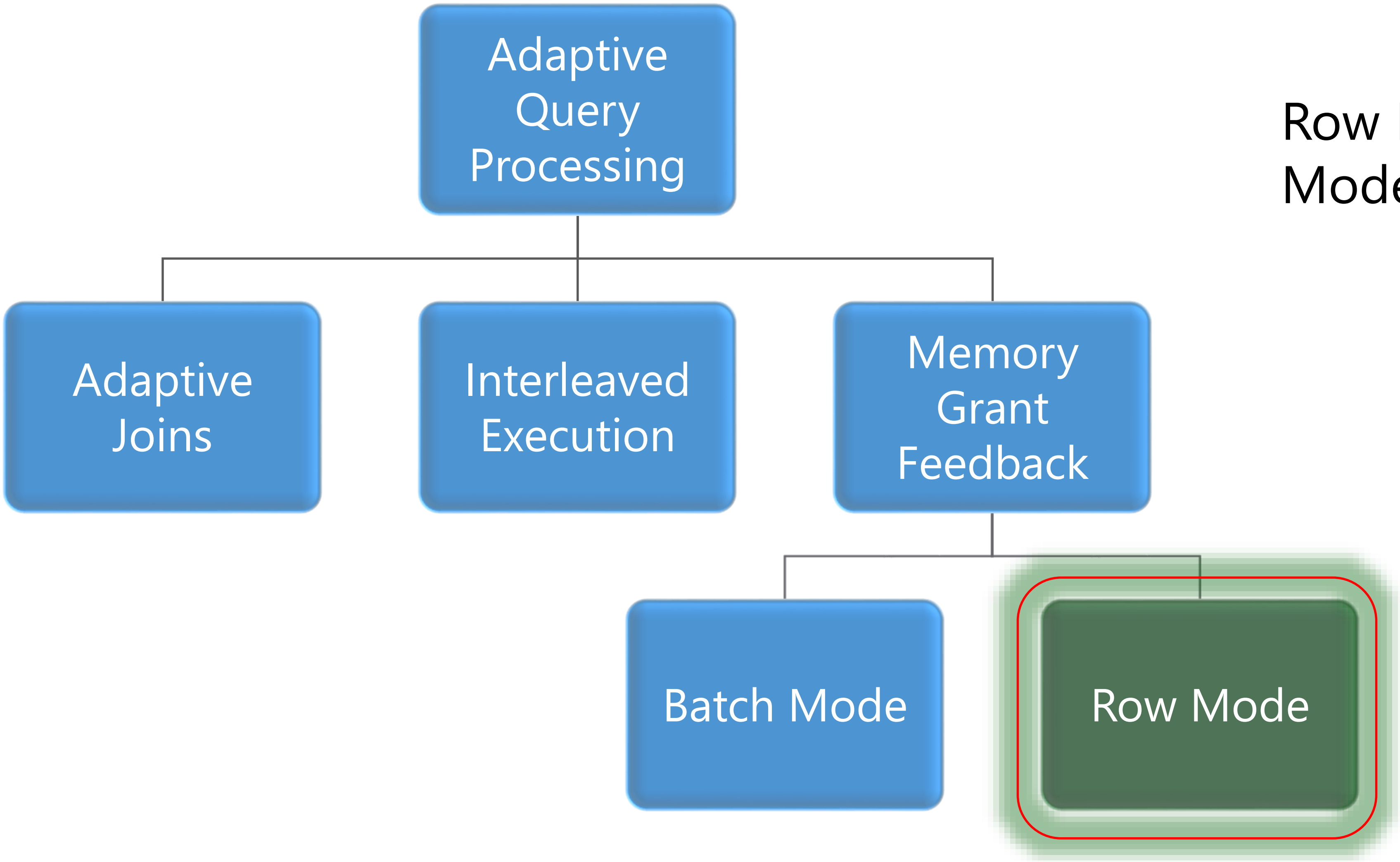


Second Execution (Memory grant adjusted)



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)

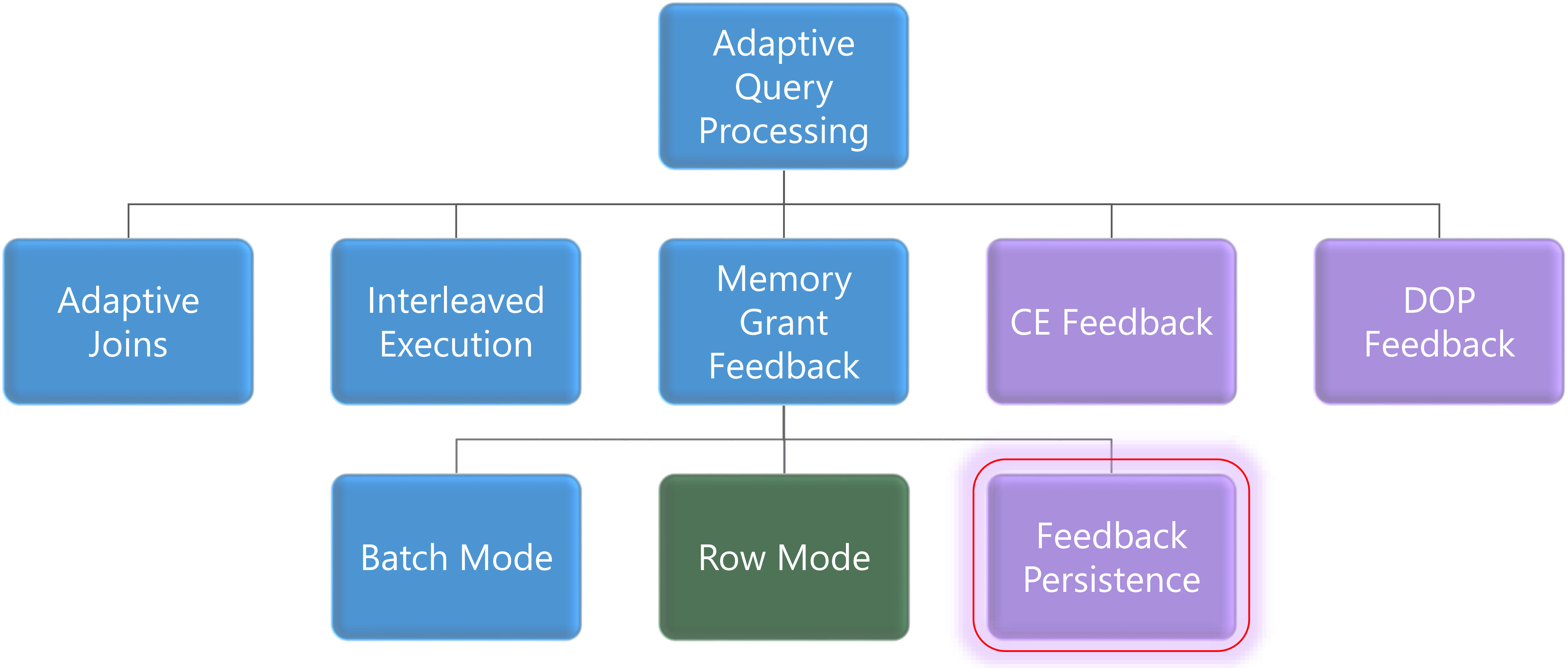
Expands on the batch mode memory grant feedback feature by also adjusting memory grant sizes for row mode operators.

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

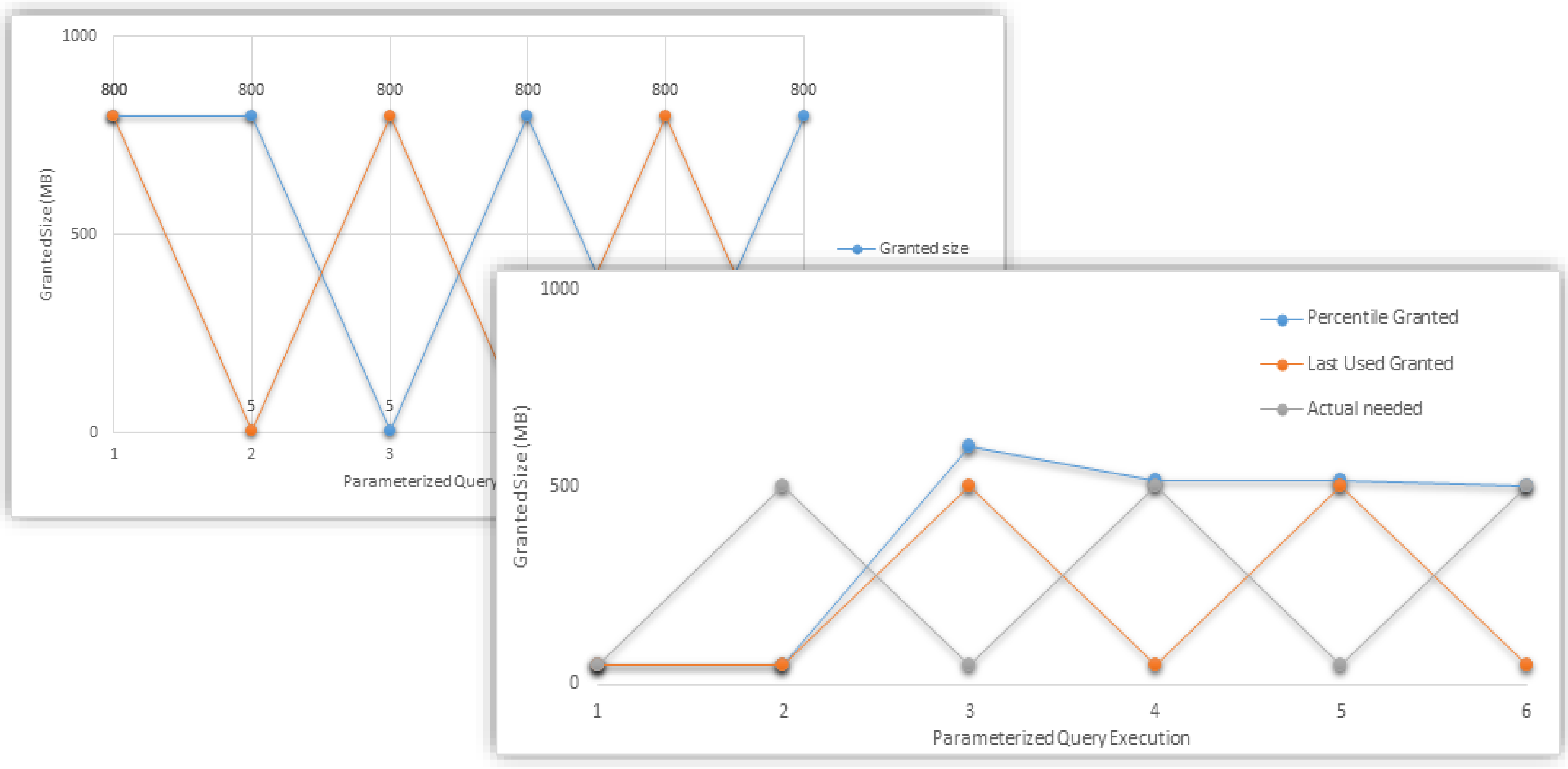
Problem: Oscillating Feedback

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

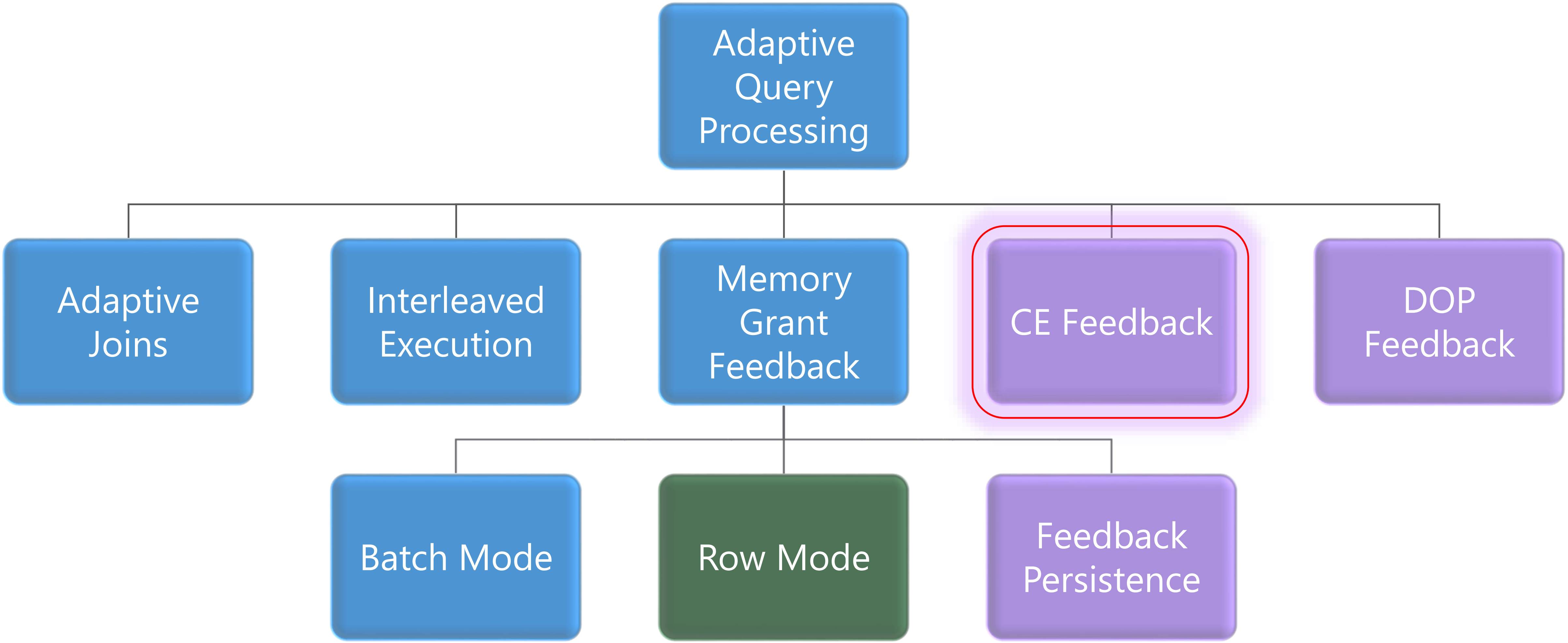
Solution: Percentile-based calculation

- 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

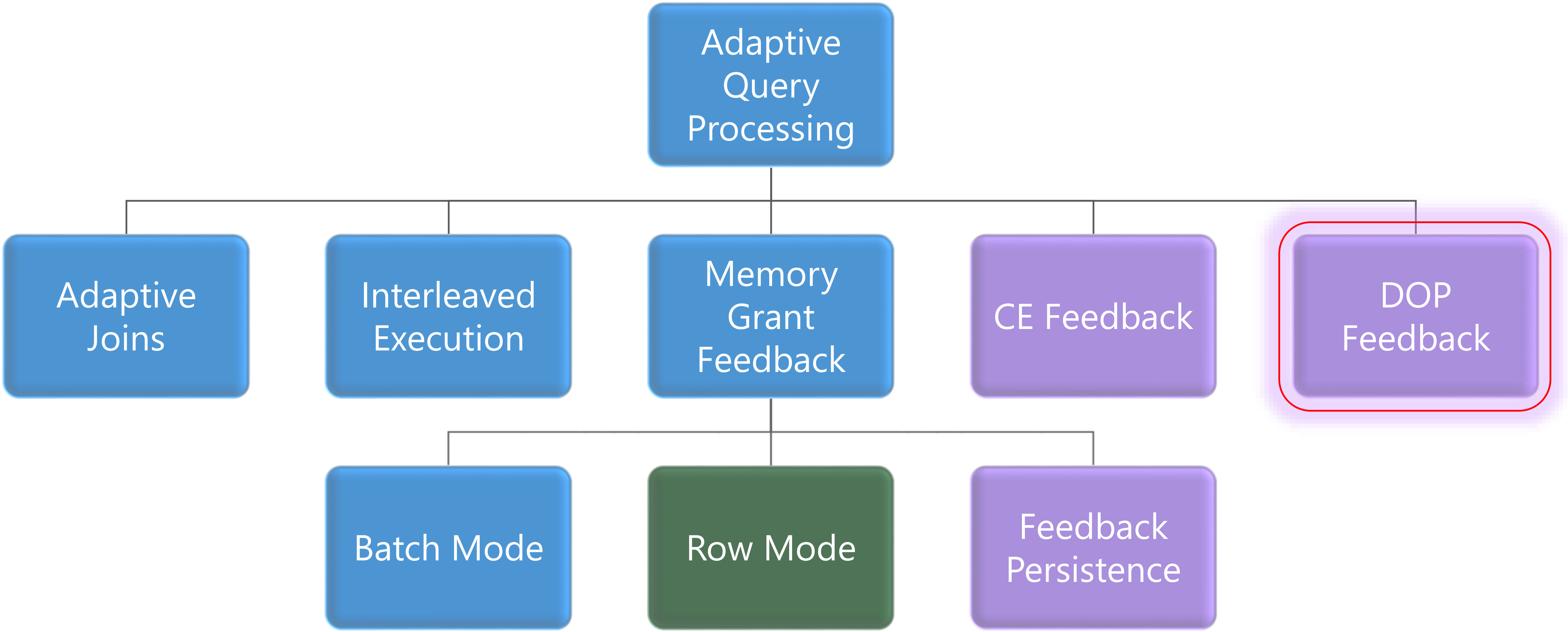
Problem: Incorrect Assumptions for Cardinality Estimates

- 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

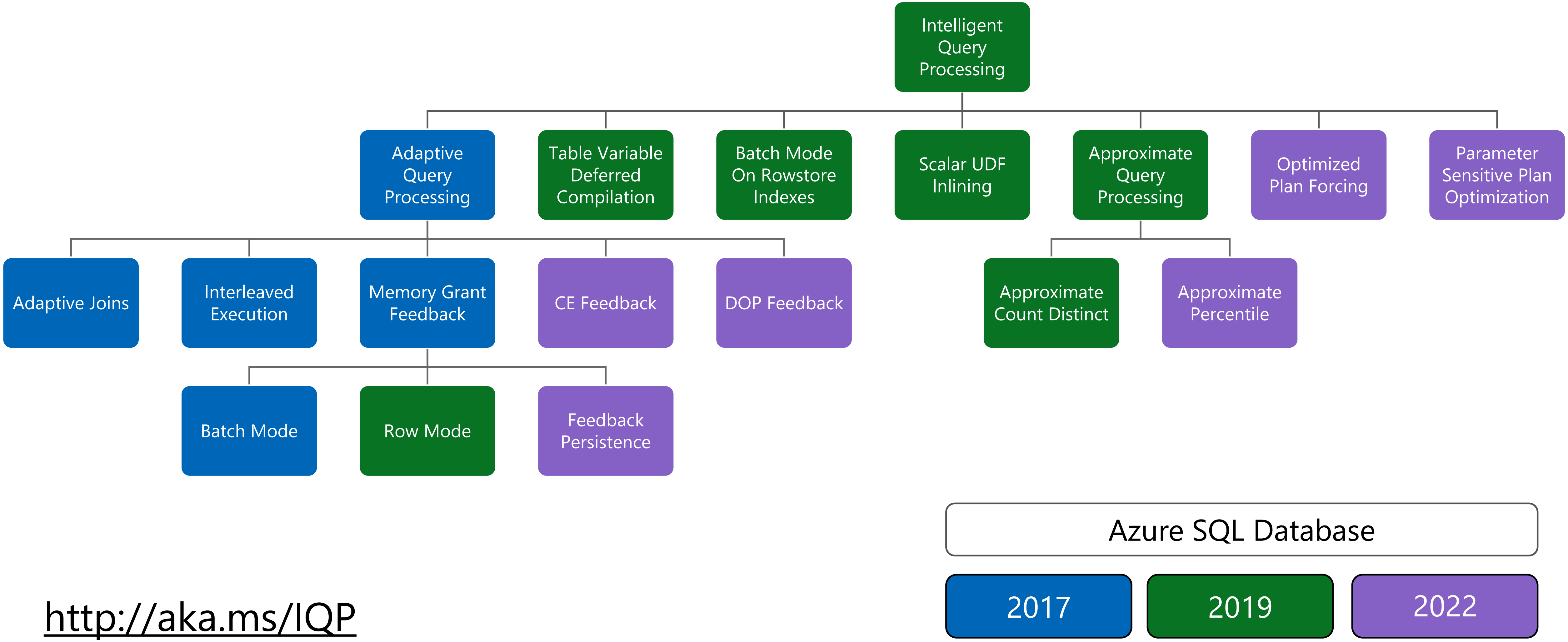
Current Settings

- 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)

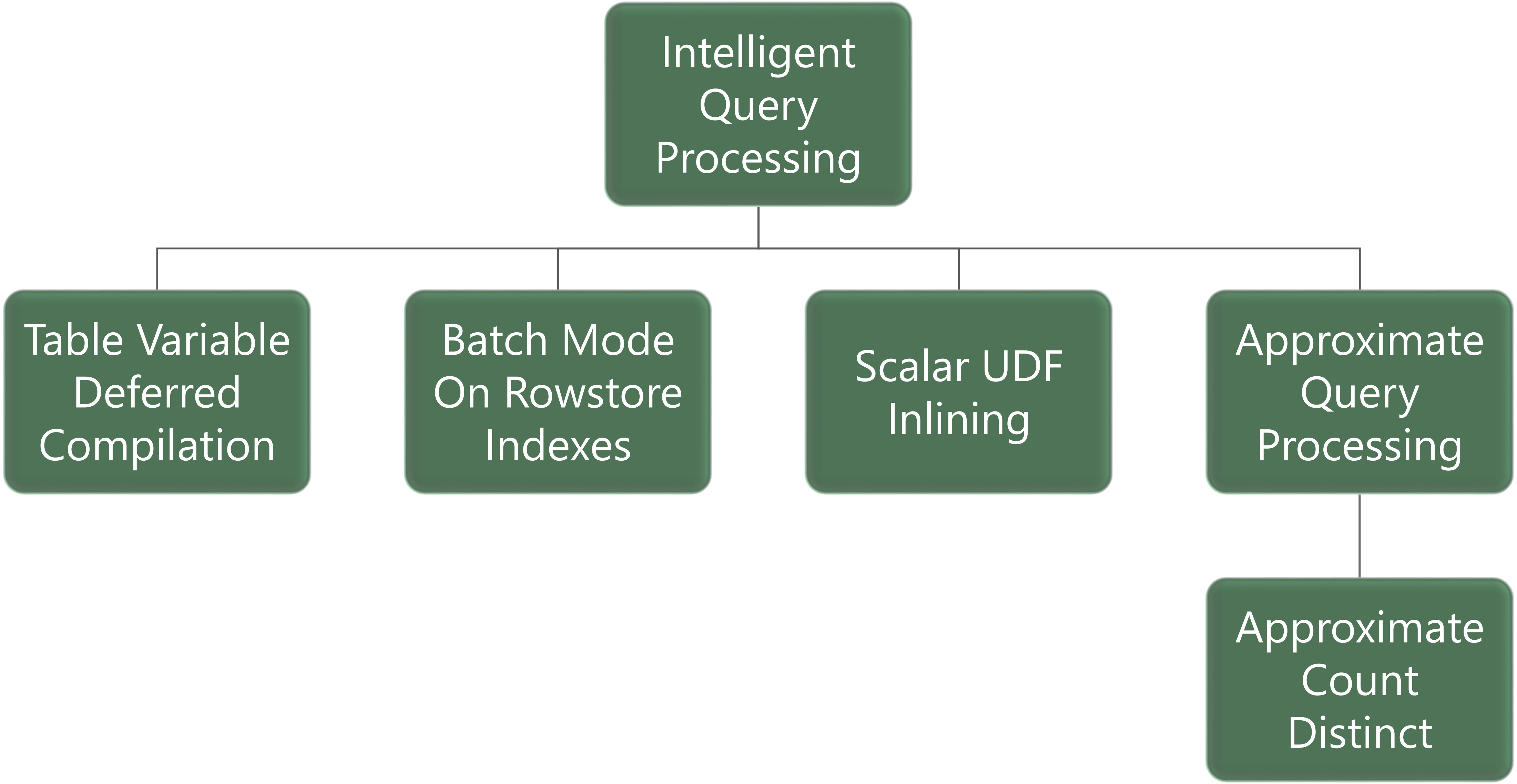
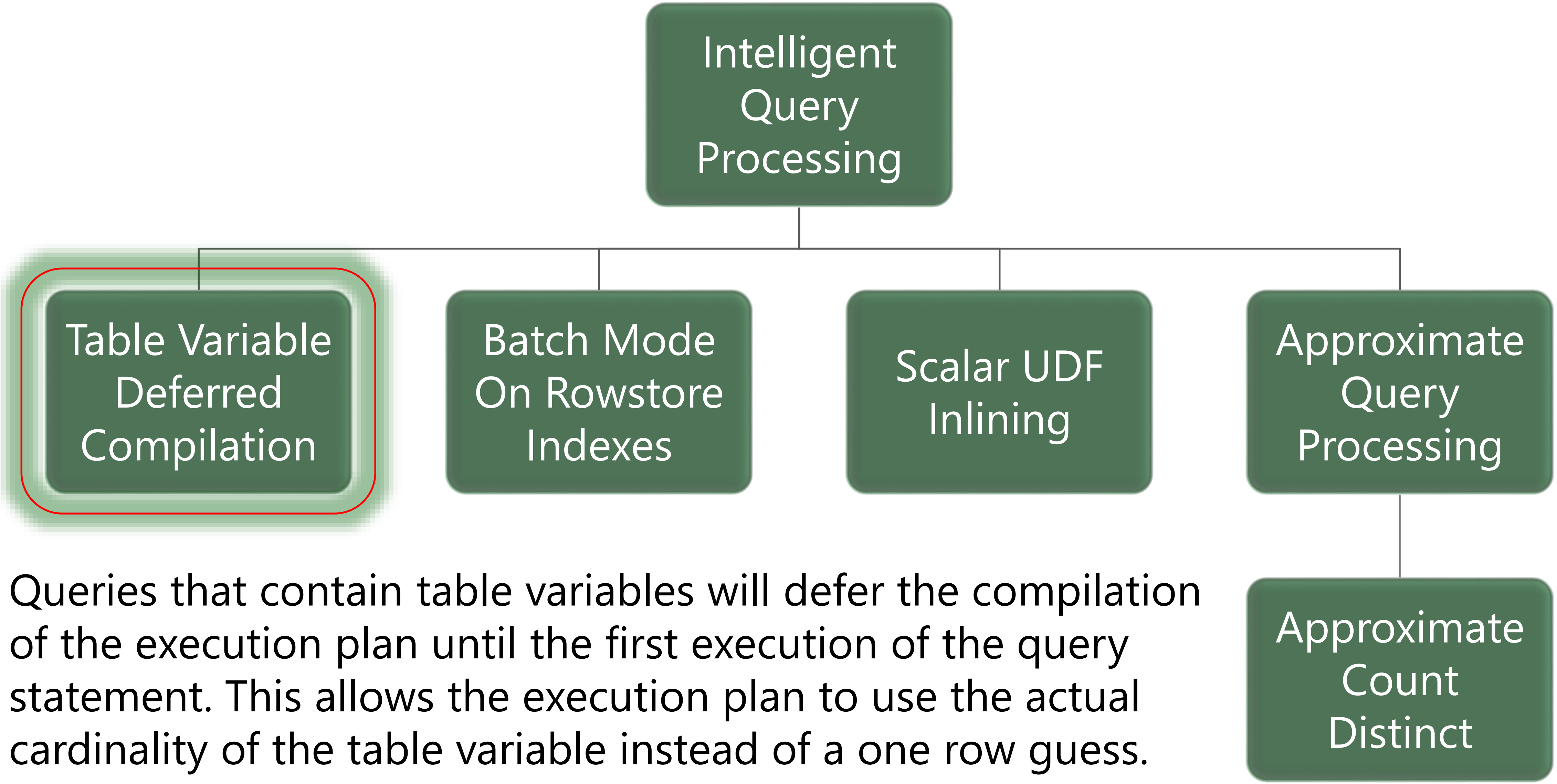


Table Variable Deferred Compilation



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

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

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

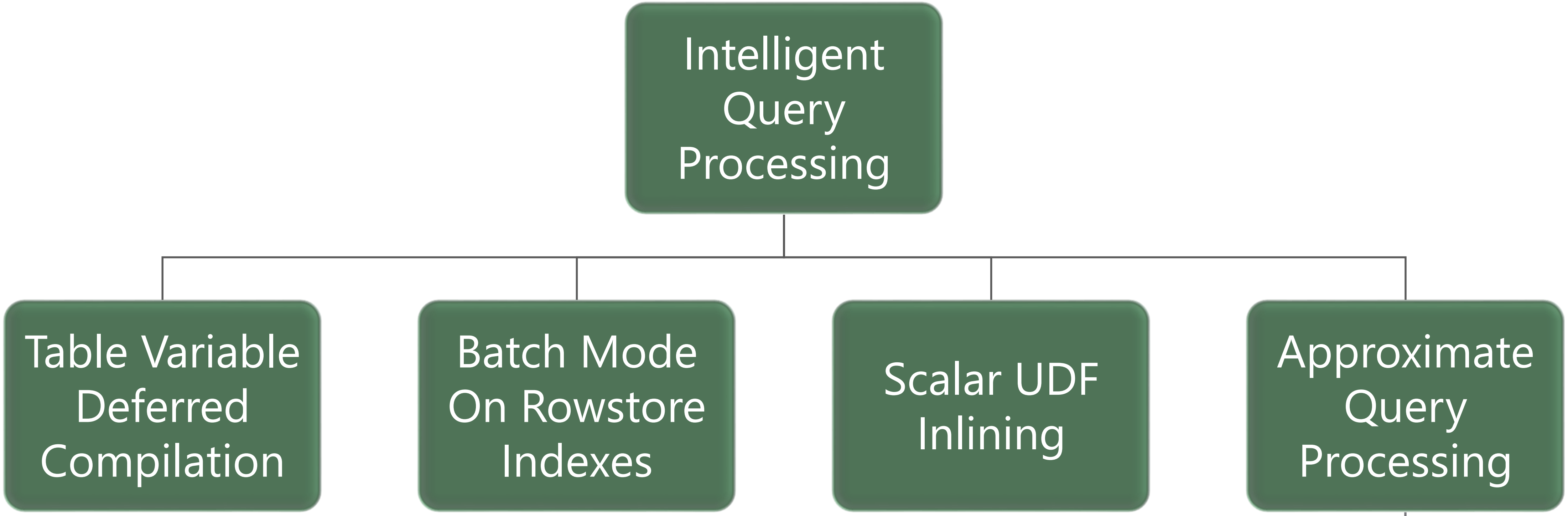
Scalar UDFs are automatically transformed into scalar expressions or scalar subqueries that are substituted in the calling query in place of the UDF operator.

Approximate Query Processing

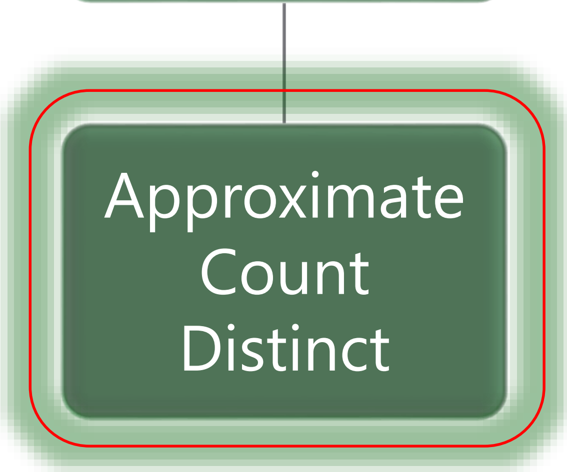
- Intelligent Query Processing
  - Table Variable Deferred Compilation
  - Batch Mode On Rowstore Indexes
  - Scalar UDF Inlining
  - Approximate Query Processing
    - Approximate Count Distinct

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

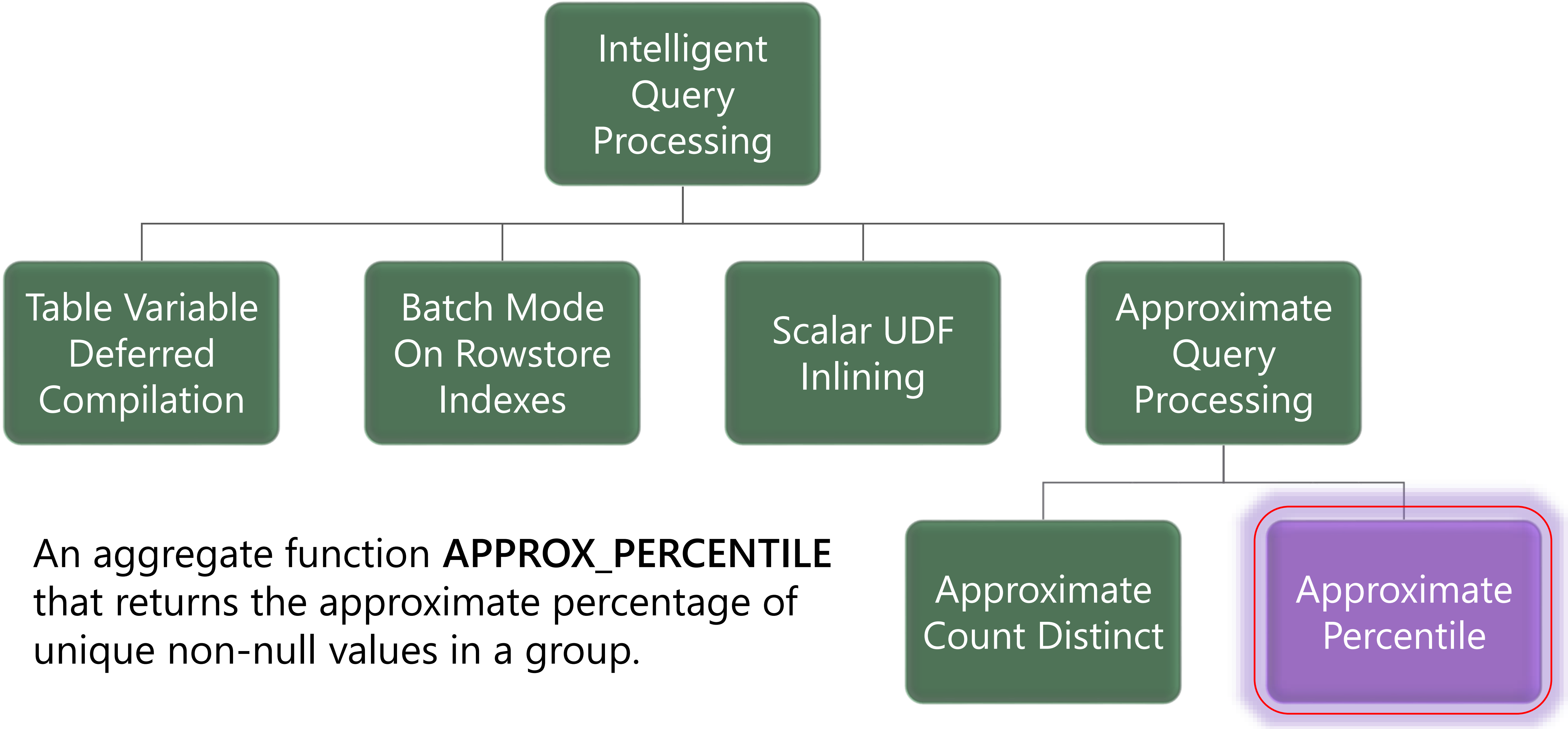
Approximate Count Distinct (2019)



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

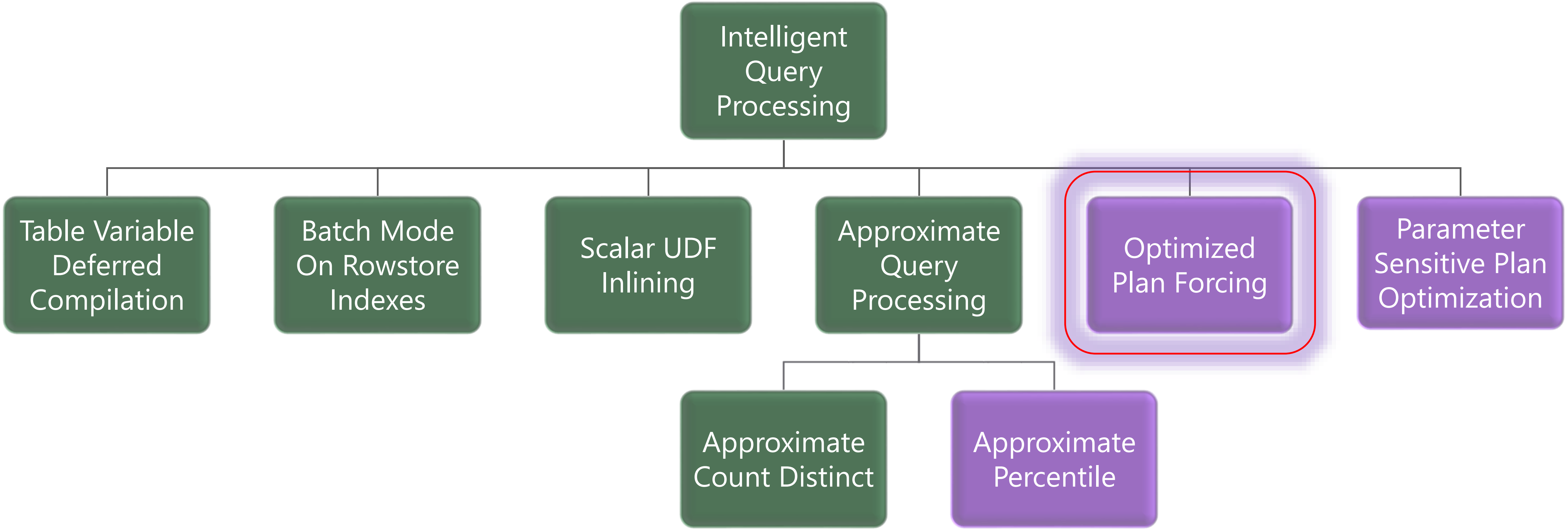


Approximate Percentile (2022)



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

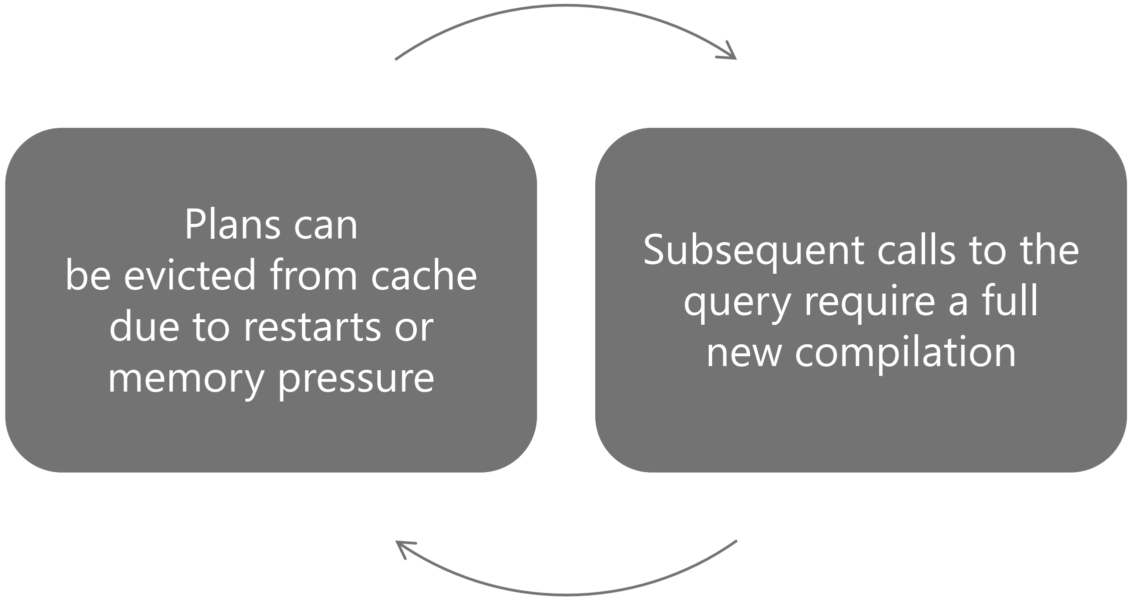
Optimized Plan Forcing (2022)



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

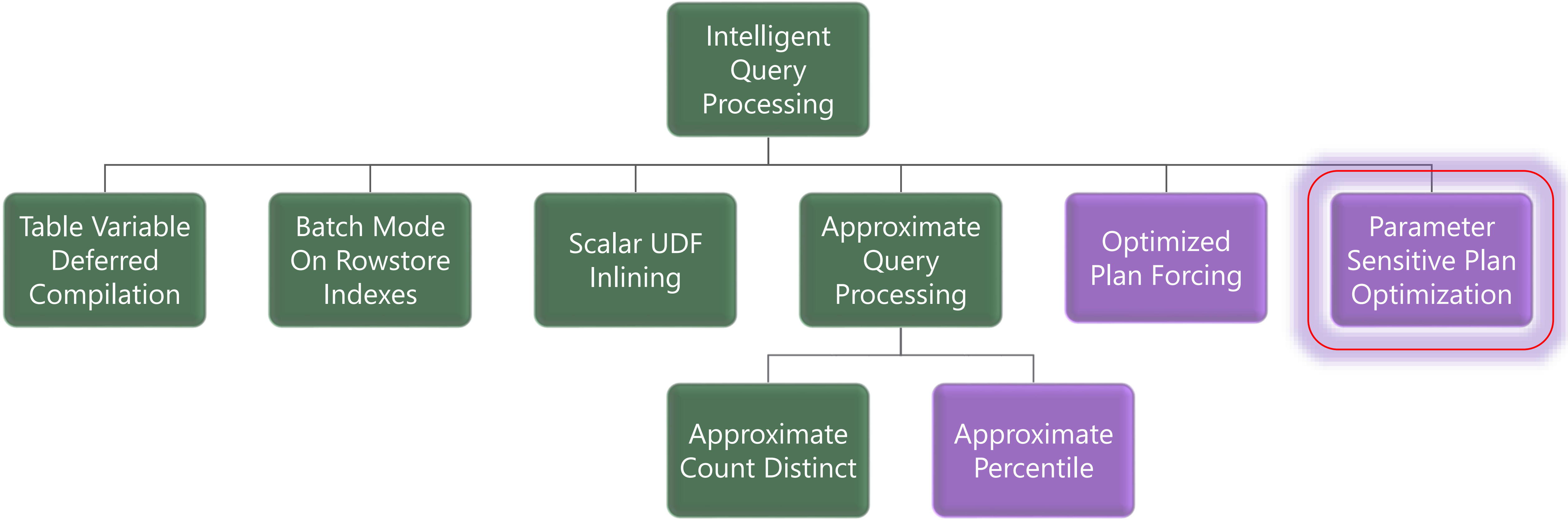


Optimized Plan Forcing (2022)

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

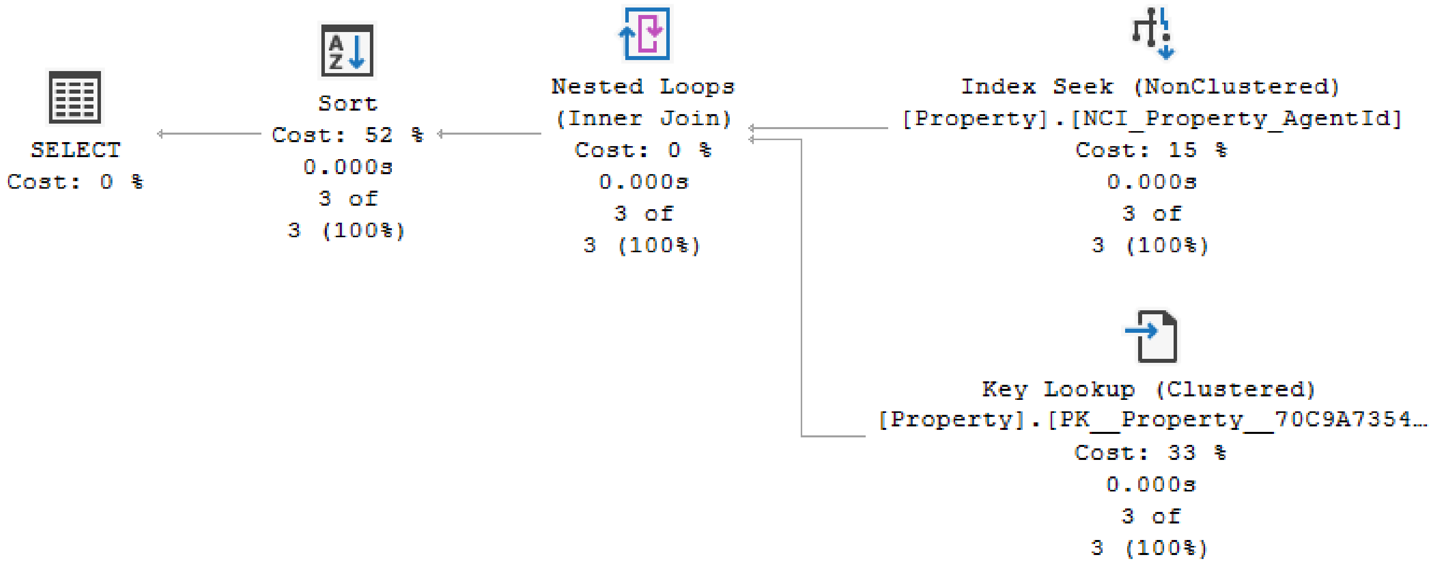
- Parameter-sniffing problem refers to a scenario where a **single** cached plan for a parameterized query is **not optimal for all** possible input parameter values
- If plan is not representative of most executions, you have a perceived “bad plan”

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

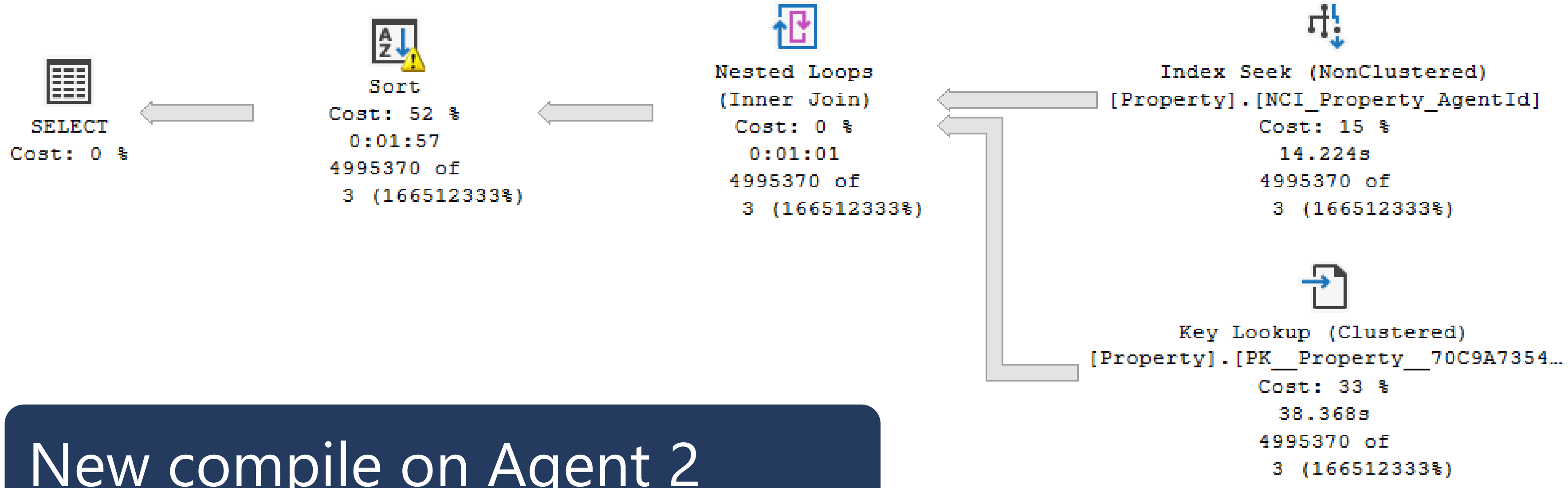


QueryTimeStats	
CpuTime	0
ElapsedTime	0

This example was borrowed from Pedro Lopes @SQLPedro

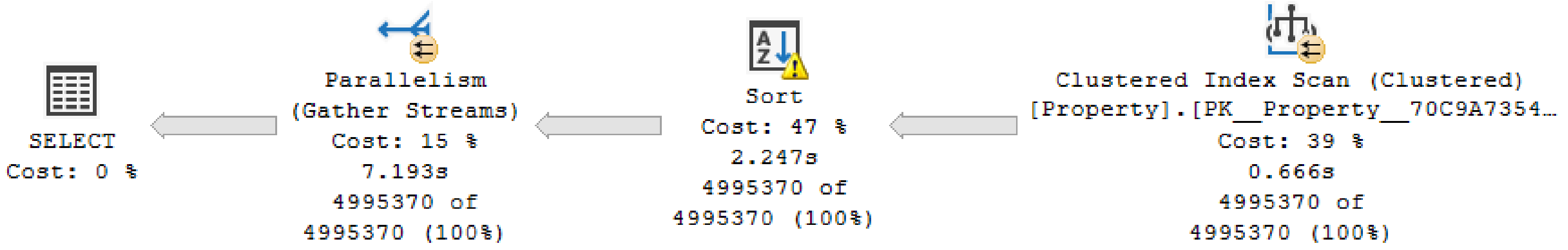
PSP today (Example of Real Estate agent's portfolio)

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

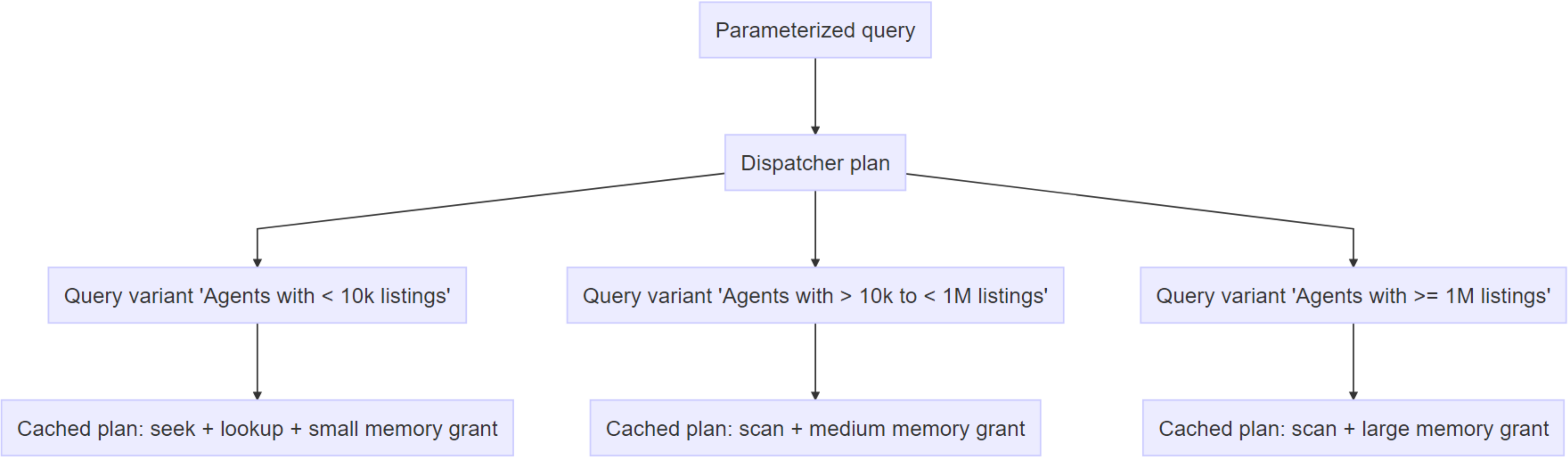
PSP Predicate Selection (2022)

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

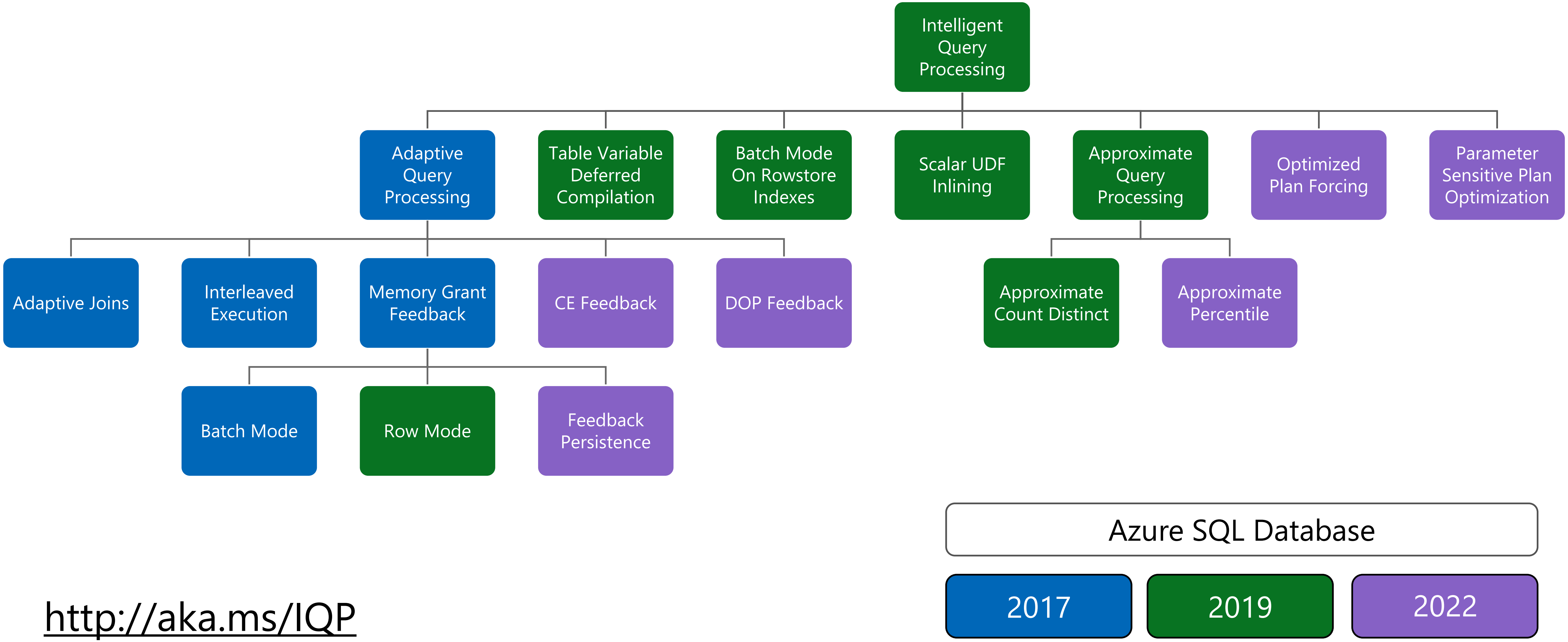
First version is scoped to equality predicates referencing statistics-covered columns; `WHERE AgentId = @AgentId`

Uses the statistics histogram to identify non-uniform distributions

Boundary Value Selection (Dispatcher Plan)



Intelligent Query Processing (2022)



http://aka.ms/IQP

Demonstration

Adaptive Query Processing (IQP)

Intelligent Query Processing – Adaptive Query Processing (IQP)

- Demonstrate APPROX_COUNT_DISTINCT

Demonstration

Intelligent query processing

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

Questions?

Knowledge Check

Explain the benefits of memory grant feedback.

Explain the benefits of batch mode on row store.

What is scalar UDF Inlining?

What Tempdb performance optimization feature is were added in SQL 2019/2022?

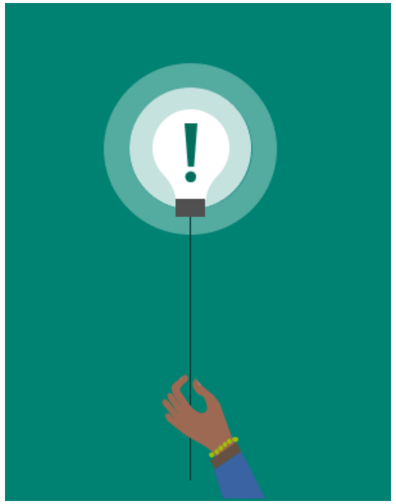
What sort of problems are solved by the improved scalability of indirect checkpoint in SQL Server starting with version 2019?

Lesson 4: Automatic Tuning

Objectives

After completing this learning, you will be able to:

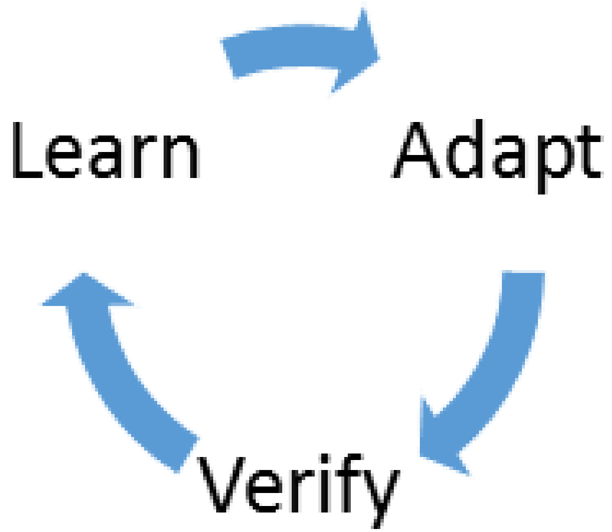
- Explain how the Automatic Tuning Feature works in SQL Server.
- Explain the limitations of the Feature.
- Describe ways to enable and disable the feature.



Automatic Tuning

Introduction

- SQL Server can be configured to collect a lot of information about database performance.
- Some performance issues can be automatically addressed by the SQL engine service.



Automatic Tuning

Automatic Plan Correction

The optimizer can choose an inefficient query plan.

- Inaccurate statistics
- Atypical parameters
- Data distribution changes

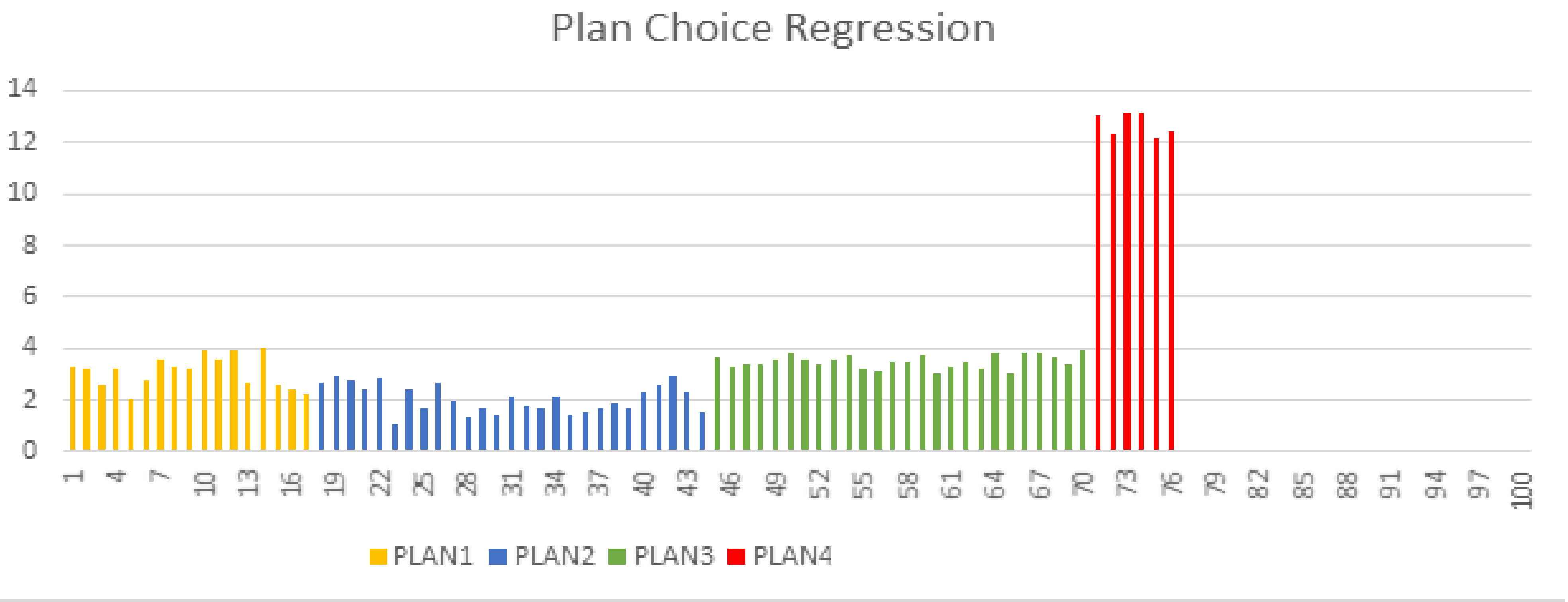
SQL Server can now track query plan performance.

If the current query plan is performing worse that previous (plan regression), SQL Server can switch to a previous plan.

Automatic Tuning

How it works

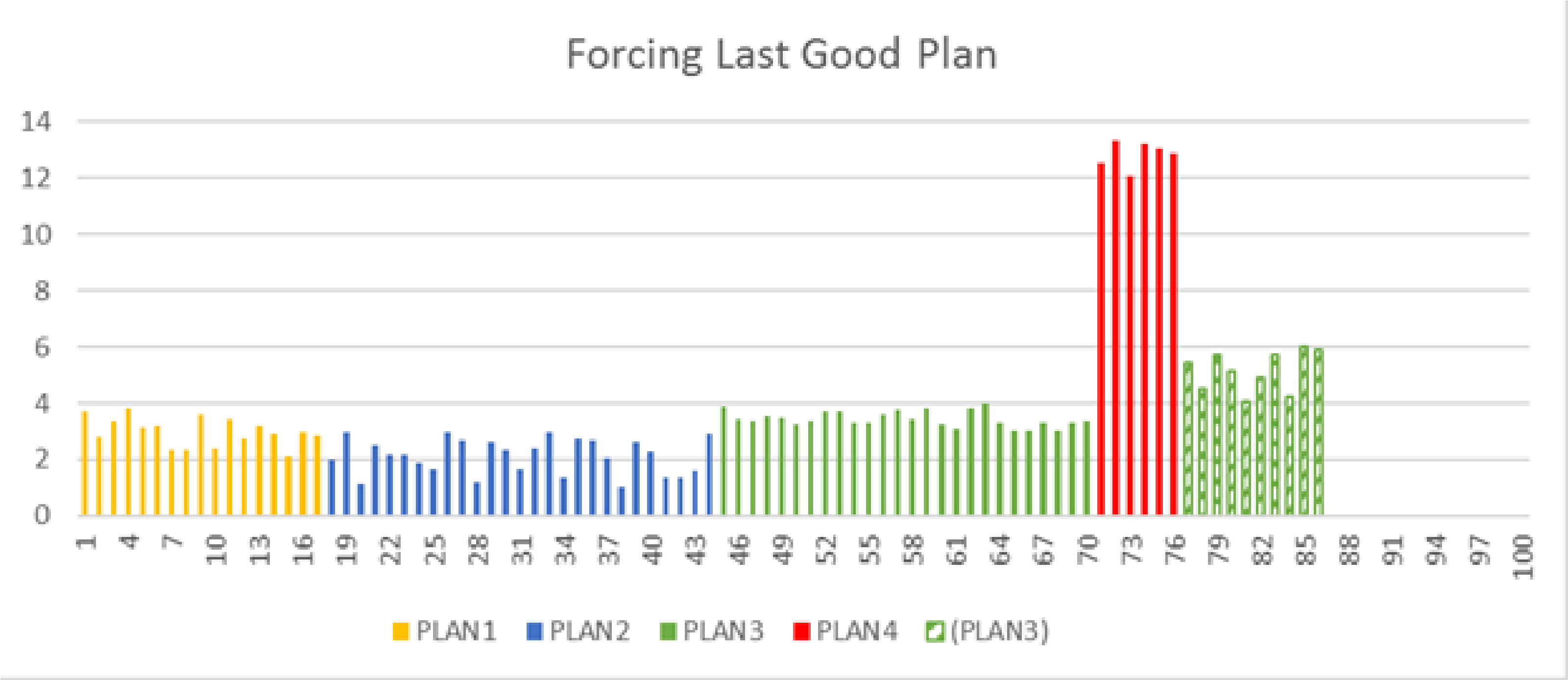
- Query Store holds performance information on queries and their query plans.
- SQL Sever will compare current query plans with the previous plan.
- Can be turned on via the Azure Portal or T-SQL.



Automatic Tuning

Last Good Plan

- If a plan is shown to be preforming significantly worse (regressed), SQL Server will attempt to force the previous plan.
- A plan is considered regressed if:
  - Uses 10 seconds of CPU more than previous plan
  - More errors than the previous plan



Automatic Tuning

Plan validation

If a plan is forced via automatic tuning

- Performance is compared against the regressed plan
- If performance is not improved, force plan will be removed

Recompiles will remove the forced plan

sys.dm_db_tuning_recommendations

- Information on why plans were chosen
- Can be used to manually apply fixes if automatic tuning is not turned on

Automatic Tuning

Considerations

Enterprise Edition only

Query Store must be turned on for the database

Automatic tuning must be enabled for the database

- ALTER DATABASE <db name> SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = ON)

Will only attempt to compare against the last plan

- May not capture queries that gradually get worse in performance

Automatic Tuning

Monitoring

After SQL Server makes an index change, the database performance is monitored.

- New indexes only kept if performance is improved.
- Dropped indexes are recreated if performance degrades.

Automatic indexes are only added when the database has enough resources to complete the action.

Automatic Tuning

Automatic Index Management

- Azure SQL Database only
- SQL Server analyzes current index usage and missing index notifications from the optimizer
- Can automatically add missing indexes
- Remove redundant or unused indexes

Demonstration

Query Store and Automatic Tuning

Questions?

Knowledge Check

Which feature needs to be enabled prior to enabling Automatic Tuning?

Which DMV can be used to discover plan choice regressions and recommended actions?

True/False? Automatic Index Management feature is available in SQL Server 2019?

Under what condition will the SQL Server engine automatically force any recommendation from Automatic Tuning Feature?

Microsoft