SQL Server Performance

SQL 2016 new innovations

Ivan Kosyakov
Microsoft Technology Center, New York
## Mission-critical performance

<table>
<thead>
<tr>
<th>Performance</th>
<th>Security</th>
<th>Availability</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In-Memory OLTP</strong></td>
<td><strong>Always Encrypted</strong></td>
<td><strong>Basic Availability Groups</strong></td>
<td><strong>Windows Server support</strong></td>
</tr>
<tr>
<td>enhancements</td>
<td>Sensitive data remains encrypted at all times, with ability to query</td>
<td>With SQL 2016 Standard Edition</td>
<td>Support for Windows Server Core and Windows Server ReFS</td>
</tr>
<tr>
<td></td>
<td><strong>Dynamic Data Masking</strong></td>
<td><strong>Enhanced AlwaysOn</strong></td>
<td><strong>Live migration</strong></td>
</tr>
<tr>
<td></td>
<td>Real-time obfuscation of data to prevent unauthorized access</td>
<td>Distributed availability groups, automatic replica seeding, distributed transactions, automatic failover, load balancing, manageability</td>
<td>Faster live migration, live migration for non-clustered VMs</td>
</tr>
<tr>
<td></td>
<td><strong>Row-Level Security</strong></td>
<td><strong>Backup enhancements</strong></td>
<td><strong>Scalability enhancements</strong></td>
</tr>
<tr>
<td></td>
<td>Fine-grained access control for table rows</td>
<td>Managed backup to Azure, Database Recovery Advisor</td>
<td>Hardware acceleration for TDE, parallelized decryption, TempDB optimization, and more</td>
</tr>
<tr>
<td></td>
<td><strong>Other enhancements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Audit success/failure of database operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TDE support for storage of In-Memory OLTP tables</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enhanced auditing for OLTP with ability to track history of record changes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Operational Analytics</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Insights on operational data; works with In-Memory OLTP and disk-based OLTP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Query Store</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monitored, optimized query plans</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Temporal Tables</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Query data as points in time and recover from accidental data changes and application errors</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In-Memory OLTP enhancements
In-memory OLTP Architecture

- No improvements in communication stack, parameter passing, result set generation
- 10-30x more efficient
- Reduced log bandwidth & contention. Log latency remains
- Checkpoints are background sequential IO

Client App

In-Memory OLTP Compiler

In-Memory OLTP Engine for Memory_optimized Tables & Indexes

Natively Compiled SPs and Schema

Parser, Catalog, Algebrizer, Optimizer

Proc/Plan cache for ad-hoc T-SQL and SPs

Interpreter for TSQL, query plans, expressions

Access Methods

Buffer Pool for Tables & Indexes

TDS Handler and Session Management

SQL Server.exe

Key

Existing SQL Component

In-Memory OLTP Component

Generated .dll

Memory-optimized Table Filegroup

Transaction Log

Data Filegroup
Performance and Scaling Improvements

Supports up to 2 TB of user data in durable memory optimized tables in a single database.

Multiple threads to persist memory-optimized tables

Parallel Support

• Parallel scan for memory-optimized tables and HASH indexes
• Parallel plan support for accessing memory-optimized tables
Query Surface Area in Native Modules

Disjunction (OR, NOT)

UNION and UNION ALL

SELECT DISTINCT

OUTER JOIN

Subqueries in SELECT statements (EXISTS, IN, scalar subqueries)

Nested execution (EXECUTE) of natively compiled modules

LOB types for parameters and variables.

Natively compiled inline table-valued functions (TVFs)

EXECUTE AS CALLER support

Built-in security functions

Increased support for built-in math functions
Transact-SQL

Support with memory-optimized tables for:

NULLable index key columns.

LOB types [varchar(max), nvarchar(max), and varbinary(max)]

UNIQUE indexes in memory-optimized tables.

FOREIGN KEY constraints between memory-optimized tables.

CHECK and UNIQUE constraints

Triggers (AFTER) for INSERT/UPDATE/DELETE operations.
The **ALTER TABLE** syntax is used for making changes to the table schema, as well as for adding, deleting, and rebuilding indexes.

Indexes are considered part of the table definition.

Key advantage is the ability to change the `BUCKET_COUNT` with an **ALTER INDEX** statement.
CREATE PROCEDURE [dbo].[usp_1] 
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER 
AS BEGIN ATOMIC WITH 
( 
 TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'us_english' 
 ) 
 SELECT c1, c2 from dbo.T1 
END 
GO 

ALTER PROCEDURE [dbo].[usp_1] 
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER 
AS BEGIN ATOMIC WITH 
( 
 TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'us_english' 
 ) 
 SELECT c1 from dbo.T1 
END 
GO 

You can now perform ALTER operations on natively compiled stored procedures using the ALTER PROCEDURE statement.

Use sp_recompile to recompile stored procedures on the next execution.
Transact-SQL

Full support for all Collation and Unicode Support

(var)char columns can use any code page supported by SQL Server

Character columns in index keys can use any SQL Server collation

Expressions in natively compiled modules as well as constraints on memory-optimized tables can use any SQL Server collation

Scalar User-Defined Functions for In-Memory OLTP

Create, drop, and alter natively compiled, scalar user-defined functions

Native compilation improves performance of the evaluation of UDFs in T-SQL
Cross-Feature Support

System-Versioned Temporal Tables
Query Store
Row-Level Security (RLS)
Multiple Active Result Sets (MARS)
Transparent Data Encryption (TDE)
Using multiple active result sets (MARS)

MARS simplifies application design:

Applications can have multiple default result sets open and can interleave reading from them.

Applications can execute other statements (for example, INSERT, UPDATE, DELETE, and stored procedure calls) while default result sets are open.

Set up MARS connection for memory-optimized tables using `MultipleActiveResultSets=True` in your connection string.

```
Data Source=MSSQL; Initial Catalog=AdventureWorks;
Integrated Security=SSPI;
MultipleActiveResultSets=True
```
In SQL Server 2016, the storage for memory-optimized tables will be encrypted as part of enabling TDE on the database. Simply follow the same steps as you would for a disk-based database.
Improvements in Management Studio

Lightweight performance analysis

Transaction Performance Analysis report pinpoints hotspots in the application

Generating migration checklists

Migration checklists show unsupported features used in current disk-based tables and interpreted T-SQL stored procedures

Generated checklists for all or some tables and procedures

Use GUI or PowerShell
Improved scaling

Other enhancements include:

- Multiple threads to persist memory-optimized tables
- Multi-threaded recovery
- MERGE operation
- Dynamic management view improvements to `sys.dm_db_xtp_checkpoint_stats` and `sys.dm_db_xtp_checkpoint_files`
- DBCC CHECKDB performance changed to support 1 TB database check improvement by 7x

In-Memory OLTP engine has been enhanced to scale linearly on servers up to 4 sockets
New Transaction Performance Analysis Overview report

New report replaces the need to use the Management Data Warehouse to analyze which tables and stored procedures are candidates for in-memory optimization.
Summary: In-Memory OLTP enhancements

Capability

ALTER support for memory-optimized tables
Greater Transact-SQL coverage

Benefits

Improved scaling: In-Memory OLTP engine has been enhanced to scale linearly on servers up to 4 sockets
Tooling improvements in Management Studio
MARS (multiple active result sets) support
TDE (Transparent Data Encryption)-enabled: all on-disk data files are now encrypted once TDE is enabled
Operational Analytics: disk-based and in-memory tables
Traditional operational/analytics architecture

**Key issues**

- Complex implementation
- Requires two servers (capital expenditures and operational expenditures)
- Data latency in analytics
- High demand; requires real-time analytics

**Presentation layer**

- BI analysts

**Application tier**

- IIS Server

**SQL Server**

- ETL: Hourly, daily, weekly

**SQL Server Relational DW**

**BI and analytics**

- Dashboards
- Reporting

**Database**
Minimizing data latency for analytics

Challenges

- Analytics queries are resource intensive and can cause blocking
- Minimizing impact on operational workloads
- Sub-optimal execution of analytics on relational schema

Benefits

- No data latency
- No ETL
- No separate data warehouse
Operational Analytics

The ability to run analytics queries concurrently with operational workloads using the same schema

Goals:

• Minimal impact on operational workloads with concurrent analytics
• Performance analytics for operational schema

Not a replacement for:

• Extreme analytics performance queries possible only using customized schemas (e.g. Star/Snowflake) and pre-aggregated cubes
• Data coming from non-relational sources
• Data coming from multiple relational sources requiring integrated analytics
Operational Analytics: disk-based tables
In-Memory ColumnStore

- Both memory and disk
- Built-in to core RDBMS engine

Customer Benefits:
- 10-100x faster
- Reduced design effort
- Work on customers’ existing hardware
- Easy upgrade; Easy deployment

“By using SQL Server 2012 In-Memory ColumnStore, we were able to extract about 100 million records in 2 or 3 seconds versus the 30 minutes required previously.”

- Atsuo Nakajima Asst Director, Bank of Nagoya
Traditional Storage Models
Data Stored Row-Wise: Heaps, b-Trees, Key-Value

- Relational, dimensional, map reduce
In-Memory DW Storage Model
Data Stored Column-wise

• Each page stores data from a single column
• Highly compressed
  - More data fits in memory
• Each column can be accessed independently
  - Fetch only columns needed
  - Can dramatically decrease I/O
In-Memory DW Index Structure
Row Groups & Segments

- A segment contains values for one column for a set of rows
- Segments for the same set of rows comprise a row group
- Segments are compressed
- Each segment stored in a separate LOB
- Segment is unit of transfer between disk and memory
## In-Memory DW Index

### Processing an Example

<table>
<thead>
<tr>
<th>OrderDateKey</th>
<th>ProductKey</th>
<th>StoreKey</th>
<th>RegionKey</th>
<th>Quantity</th>
<th>SalesAmount</th>
</tr>
</thead>
<tbody>
<tr>
<td>20101107</td>
<td>106</td>
<td>01</td>
<td>1</td>
<td>6</td>
<td>30.00</td>
</tr>
<tr>
<td>20101107</td>
<td>103</td>
<td>04</td>
<td>2</td>
<td>1</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>109</td>
<td>04</td>
<td>2</td>
<td>2</td>
<td>20.00</td>
</tr>
<tr>
<td>20101107</td>
<td>103</td>
<td>03</td>
<td>2</td>
<td>1</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>106</td>
<td>05</td>
<td>3</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>02</td>
<td>1</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101108</td>
<td>102</td>
<td>02</td>
<td>1</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>03</td>
<td>2</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101108</td>
<td>109</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>10.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>04</td>
<td>2</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>04</td>
<td>2</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101109</td>
<td>103</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>17.00</td>
</tr>
</tbody>
</table>
## Horizontally Partition Row Groups

<table>
<thead>
<tr>
<th>OrderDateKey</th>
<th>ProductKey</th>
<th>StoreKey</th>
<th>RegionKey</th>
<th>Quantity</th>
<th>SalesAmount</th>
</tr>
</thead>
<tbody>
<tr>
<td>20101107</td>
<td>106</td>
<td>01</td>
<td>1</td>
<td>6</td>
<td>30.00</td>
</tr>
<tr>
<td>20101107</td>
<td>103</td>
<td>04</td>
<td>2</td>
<td>1</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>109</td>
<td>04</td>
<td>2</td>
<td>2</td>
<td>20.00</td>
</tr>
<tr>
<td>20101107</td>
<td>103</td>
<td>03</td>
<td>2</td>
<td>1</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>106</td>
<td>05</td>
<td>3</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>02</td>
<td>1</td>
<td>5</td>
<td>25.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OrderDateKey</th>
<th>ProductKey</th>
<th>StoreKey</th>
<th>RegionKey</th>
<th>Quantity</th>
<th>SalesAmount</th>
</tr>
</thead>
<tbody>
<tr>
<td>20101108</td>
<td>102</td>
<td>02</td>
<td>1</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>03</td>
<td>2</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101108</td>
<td>109</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>10.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>04</td>
<td>2</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>04</td>
<td>2</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101109</td>
<td>103</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>17.00</td>
</tr>
</tbody>
</table>
## Vertical Partition Segments

<table>
<thead>
<tr>
<th>OrderDateKey</th>
<th>ProductKey</th>
<th>StoreKey</th>
<th>RegionKey</th>
<th>Quantity</th>
<th>SalesAmount</th>
</tr>
</thead>
<tbody>
<tr>
<td>20101107</td>
<td>106</td>
<td>01</td>
<td>1</td>
<td>6</td>
<td>30.00</td>
</tr>
<tr>
<td>20101107</td>
<td>103</td>
<td>04</td>
<td>2</td>
<td>1</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>109</td>
<td>04</td>
<td>2</td>
<td>2</td>
<td>20.00</td>
</tr>
<tr>
<td>20101107</td>
<td>103</td>
<td>03</td>
<td>2</td>
<td>1</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>106</td>
<td>05</td>
<td>3</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>02</td>
<td>1</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101108</td>
<td>102</td>
<td>02</td>
<td>1</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>03</td>
<td>2</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101108</td>
<td>109</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>10.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>04</td>
<td>4</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>04</td>
<td>2</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101109</td>
<td>103</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>17.00</td>
</tr>
</tbody>
</table>
Compress Each Segment*
Some Compress More than Others

<table>
<thead>
<tr>
<th>OrderDateKey</th>
<th>ProductKey</th>
<th>StoreKey</th>
<th>RegionKey</th>
<th>Quantity</th>
<th>SalesAmount</th>
</tr>
</thead>
<tbody>
<tr>
<td>20101107</td>
<td>106</td>
<td>01</td>
<td>1</td>
<td>6</td>
<td>30.00</td>
</tr>
<tr>
<td>20101107</td>
<td>106</td>
<td>04</td>
<td>2</td>
<td>1</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>106</td>
<td>04</td>
<td>2</td>
<td>2</td>
<td>20.00</td>
</tr>
<tr>
<td>20101107</td>
<td>106</td>
<td>03</td>
<td>3</td>
<td>4</td>
<td>17.00</td>
</tr>
<tr>
<td>20101107</td>
<td>106</td>
<td>05</td>
<td>1</td>
<td>1</td>
<td>20.00</td>
</tr>
<tr>
<td>20101107</td>
<td>103</td>
<td>02</td>
<td>1</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>02</td>
<td>1</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>03</td>
<td>5</td>
<td>5</td>
<td>25.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>01</td>
<td>2</td>
<td>1</td>
<td>10.00</td>
</tr>
<tr>
<td>20101109</td>
<td>103</td>
<td>01</td>
<td>2</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>20101109</td>
<td>106</td>
<td>04</td>
<td>1</td>
<td>1</td>
<td>25.00</td>
</tr>
<tr>
<td>20101108</td>
<td>106</td>
<td>01</td>
<td>1</td>
<td>7</td>
<td>17.00</td>
</tr>
</tbody>
</table>

*Encoding and reordering not shown
Fetch Only Needed Columns  
Segment Elimination

```
SELECT ProductKey, SUM(SalesAmount)  
FROM SalesTable  
WHERE OrderDateKey < 20101108
```
Fetch Only Needed Segments
Segment Elimination

```sql
SELECT ProductKey, SUM(SalesAmount) 
FROM SalesTable 
WHERE OrderDateKey < '20101108'
```
Operational Analytics with columnstore index

Key points

Create an updateable NCCI for analytics queries

Drop all other indexes that were created for analytics

No application changes

Columnstore index is maintained just like any other index

Query optimizer will choose columnstore index where needed
## Operational Analytics with columnstore index

### DML operations on OLTP workload

<table>
<thead>
<tr>
<th>Operation</th>
<th>B-tree (NCI)</th>
<th>Non-clustered columnstore index (NCCI)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>Insert row into B-tree.</td>
<td>Insert row into B-tree (delta store).</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>(a) Seek row(s) to be deleted. (b) Delete the row.</td>
<td>(a) Seek row in delta stores. (There can be multiple rows.) (b) If found, delete row. (c) If not found, insert key into delete row buffer.</td>
</tr>
<tr>
<td><strong>Update</strong></td>
<td>(a) Seek the row(s). (b) Update.</td>
<td>(a) Delete row (steps same as above). (b) Insert updated row into delta store.</td>
</tr>
</tbody>
</table>
Operational Analytics: minimizing columnstore overhead

Key points

Create columnstore only on cold data by using filtered predicate to minimize maintenance.

Analytics query accesses both columnstore and ‘hot’ data transparently.

Example:

Order Management Application: create nonclustered columnstore index where order_status = ‘SHIPPED’
Using Availability Groups instead of data warehouses

Always On Availability Group

Key points

Mission-critical operational workloads typically configured for high availability using Always On Availability Groups

You can offload analytics to readable secondary replica
Operational Analytics: In-Memory Tables
Operational Analytics: columnstore on In-Memory Tables

- No explicit delta row group
  - Rows (tail) not in columnstore stay in In-Memory OLTP table
  - No columnstore index overhead when operating on tail
  - Background task migrates rows from tail to columnstore in chunks of 1 million rows

- Deleted Rows Table (DRT) – Tracks deleted rows

- Columnstore data fully resident in memory
- Persisted together with operational data
- No application changes required
## Operational Analytics: columnstore overhead

### DML operations on In-Memory OLTP

<table>
<thead>
<tr>
<th>Operation</th>
<th>Hash or range index</th>
<th>HK-CCI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>Insert row into HK.</td>
<td>Insert row into HK.</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>(a) Seek row(s) to be deleted. (b) Delete the row.</td>
<td>(a) Seek row(s) to be deleted. (b) Delete the row in HK. (c) If row in TAIL, then return. If not, insert &lt;colstore-RID&gt; into DRT.</td>
</tr>
<tr>
<td><strong>Update</strong></td>
<td>(a) Seek the row(s) to be updated. (b) Update (delete/insert).</td>
<td>(a) Seek the row(s) to be updated. (b) Update (delete/insert) in HK. (c) If row in TAIL, then return If not, insert &lt;colstore-RID&gt; into DRT.</td>
</tr>
</tbody>
</table>
Operational Analytics: minimizing columnstore overhead

DML operations

Keep hot data only in in-memory tables
Example: data stays hot for 1 day, 1 week...

Workaround:
Use TF – 9975 to disable auto-compression
Force compression using a spec-proc
“sp_memory_optimized_cs_migration”

Analytics queries

Offload analytics to AlwaysOn readable secondary
## Summary of improvements

<table>
<thead>
<tr>
<th>Improvements</th>
<th>SQL Server 2014</th>
<th>SQL Server 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>clustered columnstore index</td>
<td>Master copy of the data (10x compression)</td>
<td>Master copy of the data (10x compression)</td>
</tr>
<tr>
<td></td>
<td>Only index supported; simplified analytics</td>
<td>Additional B-tree indexes for efficient equality, short-range searches, and PK/FK constraints</td>
</tr>
<tr>
<td></td>
<td>No PK/FK constraints</td>
<td>Locking granularity at row level using NCI index path</td>
</tr>
<tr>
<td></td>
<td>Uniqueness can be enforced through materialized views</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Locking granularity for UPDATE/DELETE at row group level</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DDL: ALTER, REBUILD, REORGANIZE</td>
<td>DDL: ALTER, REBUILD, REORGANIZE</td>
</tr>
<tr>
<td>updateable non-clustered index</td>
<td>Introduced in SQL Server 2012</td>
<td>Updateable</td>
</tr>
<tr>
<td></td>
<td>NCCI is <strong>read-only</strong>: no delete bitmap or delta store</td>
<td>Ability to mix OLTP and analytics workload</td>
</tr>
<tr>
<td></td>
<td>Optimizer will choose between NCCI and NCI(s)/CI or heap-based on the cost-based model</td>
<td>Ability to create filtered NCCI</td>
</tr>
<tr>
<td></td>
<td>Partitioning supported</td>
<td>Partitioning supported</td>
</tr>
<tr>
<td>equality and short-range queries</td>
<td>Row group elimination (when possible)</td>
<td>Optimizer can choose NCI on column C1; index points directly to row group</td>
</tr>
<tr>
<td></td>
<td>Partition-level scan (somewhat expensive)</td>
<td>No full index scan</td>
</tr>
<tr>
<td></td>
<td>Full index scan (expensive)</td>
<td>Covering NCI index</td>
</tr>
<tr>
<td>string predicate pushdown</td>
<td>Retrieve 10 million rows by converting dictionary encoded value to string</td>
<td>Apply filter on dictionary entries</td>
</tr>
<tr>
<td></td>
<td>Apply string predicate on 10 million rows</td>
<td>Find rows that refer to dictionary entries that qualify (R1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Find rows not eligible for this optimization (R2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scan returns (R1 + R2) rows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Filter node applies string predicate on (R2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Row returned by Filter node = (R1 + R2’)</td>
</tr>
</tbody>
</table>

---

**Performance**
# Support for index maintenance

<table>
<thead>
<tr>
<th>Operation</th>
<th>SQL Server 2014</th>
<th>SQL Server 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>Removing deleted rows</td>
<td>Requires index REBUILD</td>
<td>Index REORGANIZE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove deleted rows from single compressed RG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Merge one or more compressed RGs with deleted rows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Done ONLINE</td>
</tr>
<tr>
<td>Smaller RG size resulting from:</td>
<td>Index REBUILD</td>
<td>Index REORGANIZE</td>
</tr>
<tr>
<td>Smaller BATCHSIZE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory pressure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index build residual</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ordering rows</td>
<td>Create clustered index</td>
<td>No changes</td>
</tr>
<tr>
<td></td>
<td>Create columnstore index by dropping clustered index</td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td>Row group granularity</td>
<td>Support of SI and RCSI (non-blocking)</td>
</tr>
<tr>
<td></td>
<td>No support for RCSI or SI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Recommendation: use read uncommitted</td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td>Lock at row level (trickle insert)</td>
<td>No changes</td>
</tr>
<tr>
<td></td>
<td>Row group level for set of rows</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>Lock at row group level</td>
<td>Row-level lock in conjunction with NCI</td>
</tr>
<tr>
<td>Update</td>
<td>Lock at row group level</td>
<td>Row-level lock in conjunction with NCI</td>
</tr>
<tr>
<td></td>
<td>Implemented as Delete/Insert</td>
<td></td>
</tr>
<tr>
<td>AlwaysOn Failover Clustering (FCI)</td>
<td>Fully supported</td>
<td>Fully supported</td>
</tr>
<tr>
<td>AlwaysON Availability Groups</td>
<td>Fully supported except readable secondary</td>
<td>Fully supported with readable secondary</td>
</tr>
<tr>
<td>Index create/rebuild</td>
<td>Offline</td>
<td>Offline</td>
</tr>
</tbody>
</table>
Summary: Operational Analytics

**Capability**

Ability to run analytics queries concurrently with operational workloads using the same schema.

Data Warehouse queries can be run on In-Memory OLTP workloads with no application changes.

**Benefits**

- Minimal impact on OLTP workloads.
- Best performance and scalability available.
- Offloading analytics workload to readable secondary.

**Diagram:**

- Presentation layer
- Application tier
- SQL Server
- Database
- BI and analytics (Dashboards, Reporting)
- Analysis Services

The diagram shows the integration of BI analysts with the operational layers, highlighting the ability to run analytics queries concurrently. It also emphasizes the benefits of minimal impact and improved performance.
Query Store

Your flight data recorder for your database
Problems with query performance

Fixing query plan choice regressions is difficult
- Query plan cache is not well-suited for performance troubleshooting

Long time to detect the issue (TTD)
- Which query is slow? Why is it slow?
- What was the previous plan?

Long time to mitigate (TTM)
- Can I modify the query?
- How to use plan guide?

Plan choice change can cause these problems
The solution: Query Store

Dedicated store for query workload performance data
- Captures the history of plans for each query
- Captures the performance of each plan over time
- Persists the data to disk (works across restarts, upgrades, and recompiles)

Significantly reduces TTD/TTM
- Find regressions and other issues in seconds
- Allows you to force previous plans from history

DBA is now in control
Query Store Architecture

- Durability latency controlled by DB option
  - DATA_FLUSH_INTERNAL_SECONDS

Query Store

- Compile
  - Compile MSG
- Execute
  - Execute MSG

- Plan store
- Runtime stats
- Async Write-Back

- Query Store schema

Collections query texts (plus all relevant properties)
- Stores all plan choices and performance metrics
- Works across restarts / upgrades / recompiles
- Dramatically lowers the bar for performance troubleshooting
- New Views
- Intuitive and easy plan forcing
Query Store write architecture

Query Store captures data in-memory to minimize I/O overhead.
Data is persisted to disk asynchronously in the background.
Query Store read architecture

Views merge in-memory and on-disk content
Users always see ‘latest’ data
Query Store schema explained

Internal tables

- Query text
- Query
- Plan
- Runtime stats

Runtime stats:
- query_store_runtime_stats
- query_store_runtime_stats_interval

Compile stats:
- query_store_query_text
- query_context_settings
- query_store_query
- query_store_plan

Exposed views

sys.

- One row per query text per plan affecting option (example: ANSI NULLS on/off)
- One row per plan (for each query)
- One row per plan per time interval (example: 5 min)
Keeping stability while upgrading to SQL Server 2016

**SQL Server 2016**
Query Optimizer (QO) enhancements tied to database compatibility level

1. Install bits
2. Keep *existing* compat. level
3. Run Query Store
   (create a baseline)
4. Move to vNext CompatLevel
5. Fix regressions with plan forcing
Monitoring performance by using the Query Store

The Query Store feature provides DBAs with insight on query plan choice and performance.
Working with Query Store

/* (6) Performance analysis using Query Store views*/
SELECT q.query_id, qt.query_text_id, qt.query_sql_text,
SUM(rs.count_executions) AS total_execution_count
FROM
sys.query_store_query_text qt JOIN
sys.query_store_query q ON qt.query_text_id =
q.query_text_id JOIN
sys.query_store_plan p ON q.query_id = p.query_id JOIN
sys.query_store_runtime_stats rs ON p.plan_id = rs.plan_id
GROUP BY q.query_id, qt.query_text_id, qt.query_sql_text
ORDER BY total_execution_count DESC

/* (7) Force plan for a given query */
exec sp_query_store_force_plan
12 /*@query_id*/, 14 /*@plan_id*/
Live query statistics

View CPU/memory usage, execution time, query progress, and more

Enables rapid identification of potential bottlenecks for troubleshooting query performance issues

Allows drill down to live operator level statistics:

- Number of generated rows
- Elapsed time
- Operator progress
- Live warnings
Summary: Query Store

**Capability**
Query Store helps customers quickly find and fix query performance issues
Query Store is a ‘flight data recorder’ for database workloads

**Benefits**
Greatly simplifies query performance troubleshooting
Provides performance stability across SQL Server upgrades
Allows deeper insight into workload performance
Temporal Tables

Query back in time
Why temporal

Data changes over time

- Tracking and analyzing changes is often important

Temporal in DB

- Automatically tracks history of data changes
- Enables easy querying of historical data states

Advantages over workarounds

- Simplifies app development and maintenance
- Efficiently handles complex logic in DB engine
How to start with temporal

- ANSI 2011 compliant
- No change in programming model
- New Insights

**DDL**
- CREATE temporal TABLE PERIOD FOR SYSTEM_TIME...
- ALTER regular_table TABLE ADD PERIOD...

**DML**
- INSERT / BULK INSERT
- UPDATE
- DELETE
- MERGE

**Querying**
- SELECT * FROM temporal

**Temporal Querying**
- FOR SYSTEM_TIME AS OF FROM..TO BETWEEN..AND CONTAINED IN

Performance
Temporal database support: BETWEEN

Provides correct information about stored facts at any point in time, or between two points in time

There are two orthogonal sets of scenarios with regards to temporal data:

- System (transaction)-time
- Application-time

```
SELECT * FROM Person.BusinessEntityContact
FOR SYSTEM_TIME BETWEEN @Start AND @End
WHERE ContactTypeID = 17
```
How does system-time work?

Temporal table (actual data)  

History table

* Old versions

Update */ Delete *
Insert / Bulk Insert
How does system-time work?

Temporal table (actual data)

History table

Regular queries (current data)

Temporal queries * (Time travel, etc.)

* Include historical version
**Application-time temporal**

```sql
SELECT * FROM Employee
WHERE VALID_TIME CONTAINS '2013-06-30'
```

```sql
SELECT * FROM Employee
WHERE EmployeeNumber = 1 AND
VALID_TIME OVERLAPS PERIOD ('2013-06-30', '2014-01-01')
```

/* Temporal join */
```sql
SELECT * FROM Employee E
JOIN Position D ON E.Position = D.Position AND
D.VALID_TIME CONTAINS PERIOD E.VALID_TIME
```

**Limits of system-time**
- Time flows ‘forward only’
- System-time ≠ business-time (sometimes)
- Immutable history, future does not exist

**App-time = new scenarios**
- Correct past records as new info is available (HR, CRM, insurance, banking)
- Project future events (budgeting, what-if, loan repayment schedule)
- Batch DW loading (with delay)
Temporal data continuum
In-Memory OLTP and temporal

Extreme OLTP with cost-effective data history

Disk-based history table
Super-fast DML and current data querying
Temporal querying in interop mode

Fast DML
Internal data retention

Current data (in-memory)
In-memory buffer (internal)
Historical data (disk-based)
Summary: Temporal Tables

Quickly add historical versioning with minimal developer effort

Add temporal data to existing tables without downstream impact

Support for temporal queries, auditing, and change tracking