

# A PURE PRIMER: **OVERCOMING SQL SERVER STORAGE CHALLENGES**

Understanding the Explosion in Data and  
what Actions You Can Take



# ACKNOWLEDGEMENT



## **CONTRIBUTED AUTHOR:**

David Klee is a Microsoft MVP and VMware vExpert with a lifelong passion for the convergence of infrastructure, cloud, and database technologies. David spends his days handling performance and HA/DR architecture of mission-critical SQL Servers as the Founder of Heraflux Technologies. His areas of expertise are virtualization and performance, datacenter architecture, and risk mitigation through high availability and disaster recovery. When he is not geeking out on technologies, David is an aspiring amateur photographer. You can read his blog at [davidklee.net](http://davidklee.net), and reach him on Twitter at [@kleegeek](https://twitter.com/kleegeek).

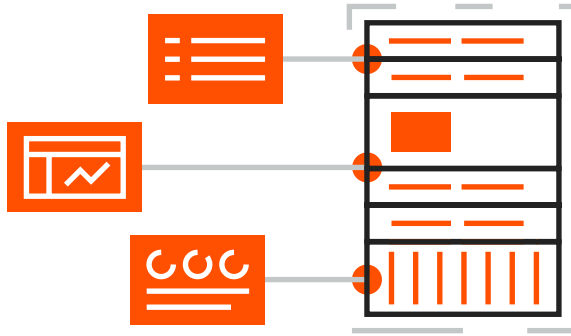
David speaks at a number of national and regional technology related events, including the PASS Summit, VMware VMworld, IT/Dev Connections, SQL Saturday events, SQL Cruise, SQL PASS virtual chapter webinars, and many SQL Server and VMware User Groups.

# TABLE OF CONTENTS

- 04 **CHAPTER 1**  
THE STATE OF DATA STORAGE
- 09 **CHAPTER 2**  
COMMON SQL SERVER BOTTLENECKS
- 19 **CHAPTER 3**  
OPTIMIZING OLTP PERFORMANCE
- 35 **CHAPTER 4**  
CREATING EFFICIENCY WITH  
COLUMNSTORE
- 46 **CHAPTER 5**  
AN INTEGRATED APPROACH TO  
MANAGING DATA PERFORMANCE

## CHAPTER 1

# THE STATE OF DATA STORAGE



Companies are retaining data in exponentially greater quantities than ever before, and organizations are scrambling to adapt and accommodate the growth.

Microsoft SQL Server and other large DBMS systems consume compute and infrastructure resources at rates rarely seen in other applications. As such, infrastructure teams have feared, and sometimes even resented, how these large databases can crush a given enterprise server infrastructure. As the volume of data grows inside an organization, these demands have pushed IT organizations to look for means to improve the performance of the systems that drive the business. Most these systems are the systems that either facilitate the operations of the business (transactional) or are decision making systems (analytics). In profiling these systems, most the bottlenecks to performance lie in how the data within these systems is either being stored or accessed on persistent storage.

Historically, this non-volatile storage was much cheaper than server memory, and therefore all data access operations were performed from disk. However, the price of memory has dropped and the capacity has grown, and now data technologies that can leverage large amounts of system memory and store the full quantity of data in memory to boost performance are growing in popularity.

The trouble is that these new data technologies required extensive modifications of existing applications, most of which software vendors are unable or unwilling to support. The Microsoft Corporation took a different approach.

## **MICROSOFT SQL SERVER**

Microsoft has acknowledged from the beginning of the product line that the consumption of these critical resources is an ever-increasing bottleneck to performance, and over the years and versions have worked to compensate for the limitations of infrastructure at the time.

The first versions of SQL Server made valid assumptions about the price and capacity of system memory. At the time, the cost of memory was very high, and the maximum quantities available in a single server were lower than the amount of data being stored. As a result, the original designs of SQL Server used disk-based storage to store the data, and only the immediate working set of data was kept in memory. As newer data was fetched from disk to fulfill a request, older data that was not currently being used was flushed from memory to make room for the newer data.

More recently, however, the system architectures have changed. The capacities of memory in modern servers have skyrocketed. The development of 64-bit CPU architecture removed the 4GB memory limitation on memory. Large quantities of server memory are also much less expensive than before. Some of the largest servers commercially available today can contain up to 384 physical CPU cores and 24TB of memory at prices far lower than previous generations of servers. Plus, at the speed of systems development, future systems are sure to contain many more times these figures.

The speed of addressable system memory is blazing fast. System memory performance is measured in several ways, most important of which is latency, measured in nanoseconds. Modern system memory operates between 13.5 and 15 nanoseconds latency. Small amounts of cache inside the CPU itself are even faster (down to one nanosecond). As a result, the amount of memory available, and the speed of that memory accessible by SQL Server is much greater than ever before. However, SQL Server’s core architecture contains a storage engine that is still oriented to having the data on disk and only fetching working set of data into memory for processing. It even assumes that all data is stored on disk and is not in memory. The storage medium that SQL Server stores the data on is much slower than server memory. The performance is also measured by latency, but the scale is much higher. Traditional storage arrays in modern datacenters measure the latency of the array’s performance in milliseconds. The fastest storage on the market today, nonvolatile enterprise flash storage, is much faster and is measured in microseconds of latency instead of milliseconds, but is still exponentially slower than server memory. The differences in scale are tremendous.

FRACTIONS OF A SECOND	METRIC NAME
.000 000 001	Nanosecond
0.000 001	Microsecond
0.001	Millisecond

So why not store all the SQL Server data in memory, since it is so much faster and cost effective? The primary challenge with leveraging conventional memory for storage of this data is that it is a volatile medium. If the power in the server goes out, the contents of memory are immediately lost. The data must reside on nonvolatile storage that can persist between power cycling.

Additionally, every bit of changed data which the business wishes to keep must be persisted to disk before the operation completes. This includes new data being inserted into the database and any changes to existing data. If the working set of data happens to be in memory at the time of the change, the operation of writing

the change back to disk slows the entire operation down to the effective speed of that disk. Today, most modern SQL Server environments are capped in their performance ceiling by how fast their storage underneath their data performs. Even servers with terabytes of memory are still underperforming because of the assumptions made by the SQL Server storage engine and the effective speed of the storage underneath.

## **IN-MEMORY FEATURES**

In recent SQL Server releases, Microsoft's efforts to improve the performance of the server by leveraging in-memory technologies has resulted in two key distinct features.

- In-Memory OLTP
- Columnstore Indexes

Microsoft is also committed to eliminating the exclusivity of these features. As of SQL Server 2016 Service Pack 1, In-Memory OLTP and Columnstore indexes are now available for use in SQL Server Standard Edition. Previously, these features were limited to those using Enterprise Edition only. Let's explore these exciting new features.

## **PROJECT HEKATON & IN-MEMORY OLTP**

Microsoft acknowledged that this basic assumption of storage and memory was becoming an ever-increasing bottleneck to performance, and in 2008, began work on building a database platform designed for in-memory operations. The origins of this project came from the goal to be 100 times faster than the base SQL Server engine, and the codename Hekaton, the Greek word for 100, was chosen.

Microsoft elected to embed the new engine inside the existing SQL Server engine instead of delivering a new product, and it became the SQL Server In-Memory OLTP feature that first shipped with the SQL Server 2014 release. This new engine allows developers to utilize new memory-optimized data structures and improve the performance of key business processes, all while reducing development time because the base platform is extended and not replaced.

## COLUMNSTORE INDEXES

Microsoft also realized that traditional database row-based indexes are inefficient for certain types of queries, including data warehouse queries, analytical processing, and reporting. Based on xVelocity, an advanced technology originally developed for use with PowerPivot and SQL Server Analysis Services, and beginning with the SQL Server 2012 release, developers can now define columnstore indexes on database tables. Columnstore indexes stores columns instead of rows in a memory-optimized columnar format instead of the B-tree structure of traditional indexes, and can be used to improve the efficiency of certain types of queries while reducing the memory footprint of the operation.

## UP NEXT

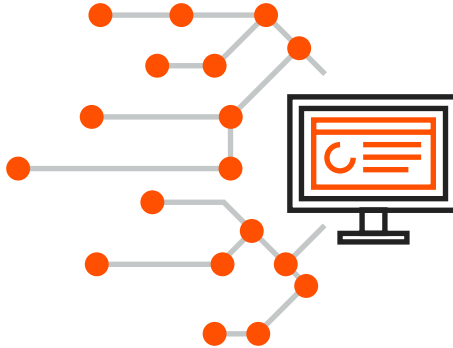
Each one of these features can lead to significant performance improvements in certain scenarios and use cases, but each come with its own implications and architectural considerations on the enterprise datacenter when stepping back and looking at the bigger picture. Database Administrators (DBAs) might not have the exposure or visibility into the infrastructure underneath the data platform, and might not see the net changes in the infrastructure, or the direct impact to the infrastructure, as a result of leveraging these technologies.

The following chapters will explore each of these features in-depth and the infrastructure underneath, present techniques and scenarios for when and how to leverage these features, and discuss the architectural challenges and considerations necessary to successfully implement these features in today's modern datacenters.



## CHAPTER 2

# COMMON SQL SERVER BOTTLENECKS

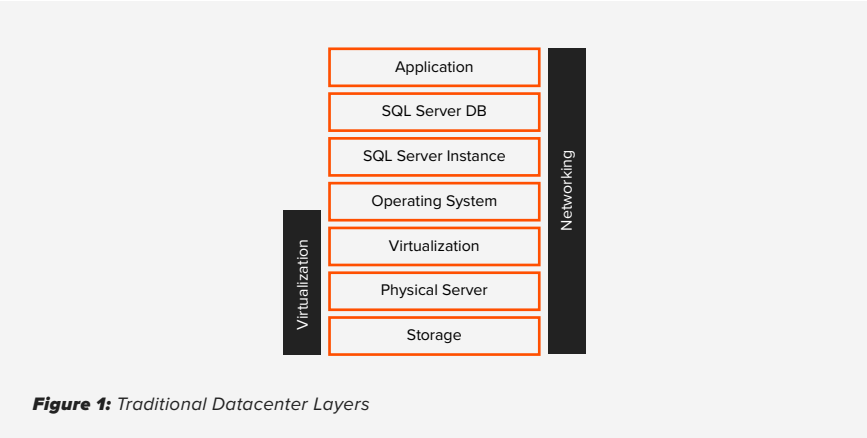


Microsoft's SQL Server engine can only perform as fast as the largest bottleneck in the entire system stack allows. In any modern datacenter, both in the cloud and on-premises, bottlenecks are sure to exist at or around the database layer. SQL Server places an incredible demand on the infrastructure, and in-memory features push certain areas even harder. If the bottlenecks to peak performance are in an area that impacts memory performance, in-memory database features could perform slower than their on-disk counterparts.

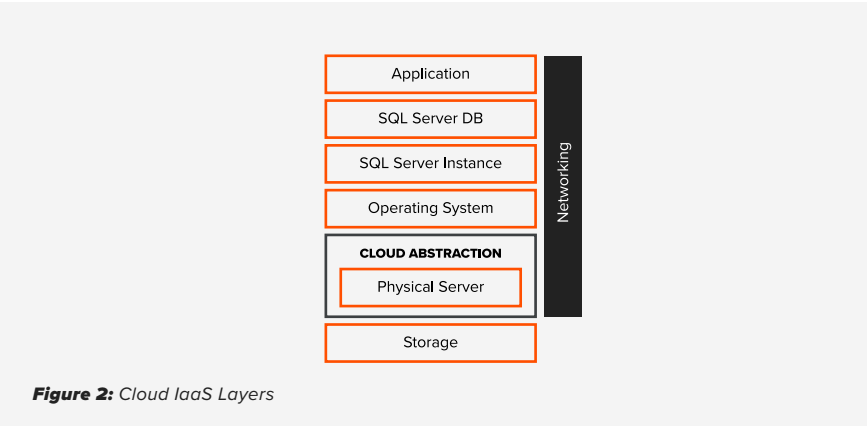
Let's look through these layers to familiarize you with them and how they impact the performance of the database server.

# INFRASTRUCTURE STACKS

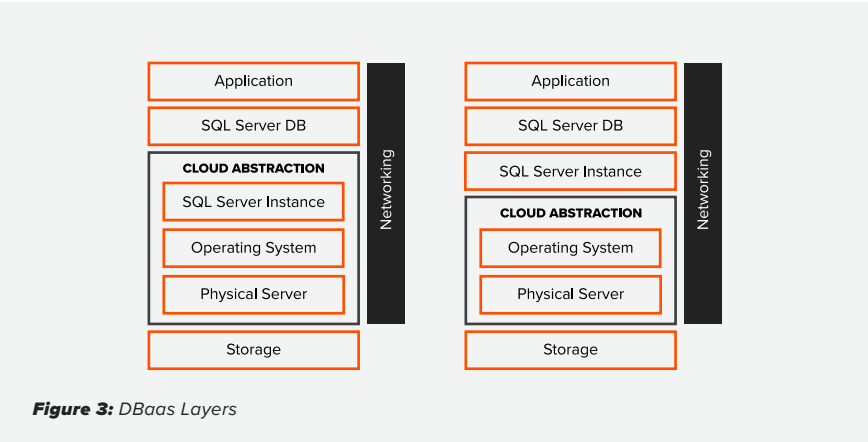
The layers around the database engine are quite distinct. When referring to the traditional datacenter model, the layers are relatively fixed, and virtualization is (now) assumed to be in use.



If the SQL Server is placed in a virtual machine in a public cloud infrastructure-as-a-service (IaaS) platform, such as Microsoft Azure or Amazon AWS, these layers are still present, but certain parts of the stack are now abstracted away from the end user.



In the IaaS model, the virtualization and the physical server layers are replaced with an automated abstraction layer that allows end users to provision VMs. Storage is connected through this abstraction layer, and the end user selects the performance class and capacity of the storage. The end user has no ability to see the specifics of the storage or hardware. If the model is pushed even further towards the database-as-a-service model (DBaaS), an even greater level of abstraction is exhibited.



In some DBaaS models (left), such as Microsoft Azure SQL Database, the abstraction layer occurs all the way up to the individual SQL Server database, and the instance layer is abstracted and not accessible. Individual databases are provisioned independently of the others. In other cases (right), such as Amazon RDS for SQL Server, the abstraction layer stops at the SQL Server instance, and the end user can provision any number of databases on the instance.

In all the cases, the layers are still there, and each have their own performance characteristics and associated performance challenges. Let's briefly explore each layer, and discuss how each relates to SQL Server and in-memory data performance.

## STORAGE

Arguably, the storage layer is the most important component of the infrastructure stack. The storage is where the most valuable asset of the business, its data, permanently resides. Many options for storage exist, no matter if the target platform is a datacenter on-premises or in the public cloud. While the vast majority of these options are highly available (and you should not use ones that are not), the performance characteristics and capacity of these platforms vary wildly between platforms and vendors. Now, if the purpose of in-memory databases is to have all the data completely in memory and not rely on disk, why is storage so important? Two primary reasons exist.

The first reason is clear. All of this data must be stored on disk if the architect wishes to maintain this data between system reboots. The architect can elect to make certain database tables not persist to disk, but this data is lost and gone forever if the database engine or operating system is restarted. For the data that is required to persist, any change to the data must be written to disk before that transaction can complete. The speed of the storage must be quite fast, with the lowest possible latency to disk, to help keep the performance of the transaction at acceptable levels. Essentially, the fastest possible storage is required for persisting in-memory data to disk, because any changes to disk now cause the in-memory operation to perform at the (much slower) speed of disk.

The second reason is less intuitive. The performance of the storage matters not just for the speed of the individual transaction, but also for system startup. For data that is to be in memory, all of it must be read from disk and loaded into memory at system startup before that data is ready for use and accessible by end users. If the storage subsystem is slow, the startup time for that database can cause the database to be unavailable for an extended period.

## PHYSICAL SERVER

The physical server is the next step up from the storage, and contains the primary compute resources of the platform – CPUs and memory. The speed and scale of these servers are, just like any other component in the infrastructure, variable

based on design, but are quite fast. Modern servers can scale from one up to sixteen CPU sockets, each with CPUs containing many processing cores, with terabytes of available memory. The speed of these platforms can vary, even within a model line. When storage is no longer the principal bottleneck within the infrastructure, the database workloads are primarily CPU and memory bound, and the speed of these two components matter more than before. CPU speeds are measured in number of cores and clock speed (measured in gigahertz, or Hz) per core. Not all CPUs are created equal, and the speed varies between CPUs. GHz is a measure of speed but not of raw performance. Benchmark metrics exist for all modern CPUs to help you select the fastest possible CPUs for your workload.

For example, based on the PassMark benchmark at the time of this writing (available at [cpubenchmark.net](http://cpubenchmark.net)), a single CPU core on an Intel Xeon E5-1680 v4 CPU running at 3.4GHz has a CPU Mark score of 2,295. AN Intel Xeon E5-2690 v4 CPU running at 2.6GHz has a CPU Mark score of 1,994, or 13% slower per thread. In many mission-critical workloads, a 13% improvement is quite substantial, and well worth the small additional price for the faster CPUs on each server. Most operations in an OLTP-based database server are not parallelized. Make your CPU selection wisely. If spending a small amount of additional capital on a server to purchase CPUs with faster core performance, even at the expense of a slightly reduced core count, the net improvement to performance should be felt by the organization.

A lesser known, but arguably more important factor to performance is the speed of the memory. Most hardware integrators to are not aware that certain memory configurations can result in degraded memory performance. A detailed diagram of this memory speed configuration can be found for a Cisco UCS B200 M4 server at <http://hfxte.ch/memspeed> on page 44 of the PDF. An abridged version of the memory configuration and corresponding speed is as follows.

SERVER MEMORY	CPU 1 DIMMS			CPU 2 DIMMS			SPEED (MHZ)	TOTAL DIMMS
	BANK 1	BANK 2	BANK 3	BANK 1	BANK 2	BANK 3		
128GB	4x8GB	4x8GB	-	4x8GB	4x8GB	-	2133	16
128GB	4x16GB	-	-	4x16GB	-	-	2133	8
192GB	4x16GB	4x8GB	-	4x16GB	4x8GB	-	2133	16
192GB	4x8GB	4x8GB	4x8GB	4x8GB	4x8GB	4x8GB	1600	24
256GB	4x16GB	4x16GB	-	4x16GB	4x16GB	-	2133	16
256GB	4x32GB	-	-	4x32GB	-	-	2133	8
384GB	4x16GB	4x8GB	4x8GB	4x16GB	4x8GB	4x8GB	1866	24
512GB	4x32GB	4x8GB	-	4x32GB	4x8GB	-	2133	16
512GB	4x64GB	-	-	4x64GB	-	-	2133	8
768GB	4x32GB	4x32GB	4x32GB	4x32GB	4x32GB	4x32GB	1866	24
1024GB	4x64GB	4x64GB	-	4x64GB	4x64GB	-	2133	16
1024GB	4x64GB	4x64GB	4x64GB	4x64GB	4x64GB	4x64GB	1600	24

**Figure 4:** Hardware Memory Configuration & Speeds

On this particular server, when the third bank of memory slots per CPU socket is filled, the memory speed can be up to 25% slower than the faster configurations. When the SQL Server is leveraging in-memory features, a 25% reduction in core memory speed, even with the incorporation of triple-channel memory access, could result in substantially degraded performance for the SQL Server.

This performance reduction is silent. It is not going to set entries in the event log or be accessible via SQL Server DMOs. The performance is reduced, and rarely to administrators notice. In short, ensure that the CPU selection and memory configuration are designed for the most optimal performance that the server budget allows.

## VIRTUALIZATION
















Virtualization is a ubiquitous technology in most modern data centers. It is a software layer, called a hypervisor, that is installed on a physical server that allows multiple compartmentalized operating systems (virtual machines, or VMs) to coexist and run on the same physical server, all while not aware that other virtual machines are on the same physical server. Common hypervisors are VMware's ESXi and Microsoft's Hyper-V. VMs are abstracted away from being dependent on a particular type and configuration of hardware, and allows for the VMs to be portable. VMs can migrate from one physical server to another with no interruption in service, and since they are no longer dependent on a single physical server, can be replicated to a disaster recovery site much easier than before. Hopefully by now all your production SQL Servers are virtualized with some modern variant of a virtualization hypervisor.

The performance overhead is so insignificant, it can hardly be measured, let alone 'felt' by the applications. The advantages to fully virtualize all servers far outweighs the few challenges with virtualization. At this point in time, the majority of the challenges are operational, rather than technical. These challenges include differences in backup ideologies between the organizational silos, high availability expectations, and other management and monitoring aspects.

## CLOUD IAAS

IaaS technologies are, for all intents and purposes, nearly identical to on-premises virtualization – at least from an architectural level. Public cloud IaaS, or VMs in the cloud, is just virtualization in someone else's datacenter with sophisticated automation and security measures placed in front of it. The automation portion of IaaS, usually a web-based portal, allows end users to provision and manage these virtual machines. Instead of purchasing hardware, storage, interconnects, and associated licensing, users pay for the compute resources and bandwidth consumed on an hourly basis. It is treated more as a utility, and can shift the IT infrastructure expenses from capital expenditures to more of an operational expense model.

Virtual machines are provisioned in the public cloud through a sizing model. Various cloud platforms have their own sizing grid, and you should have a thorough understanding of the resource consumption and allocations that your virtual machines require. Some cloud platforms are harder than others to resize and scale the virtual machine, so be careful when you size the machine. Under-sizing can cause your application to perform poorly. Oversizing the VM can cause elevated costs on the cloud platform.

DS1_V2 Standard ★		DS2_V2 Standard ★		DS3_V2 Standard	
1	Core	2	Cores	4	Cores
3.5	GB	7	GB	14	GB
	2 Data disks		4 Data disks		8 Data disks
	3200 Max IOPS		6400 Max IOPS		12800 Max IOPS
	7 GB Local SSD		14 GB Local SSD		28 GB Local SSD
	Load balancing		Load balancing		Load balancing
	Premium disk support		Premium disk support		Premium disk support

**Figure 5:** Microsoft Azure VM Sizing Example

Additional cloud-based storage can be provisioned separately, attached to these VMs, and used for additional space and/or speed as the database demand grows over time. Note that the provisioned storage contains an IOPs limitation by data disk, and careful planning should be taken to provision enough disk speed to handle the target workload.

## CLOUD DBAAS

Taking the cloud model even further, the database-as-a-service model allows end users to only focus on the database instance and databases, and eliminates the need (and even the access) to manage the operating system layer. End users can simply focus on provisioning SQL Server instances and databases (e.g. Amazon RDS for SQL Server) or provisioning databases (e.g. Microsoft Azure SQL Database). The application connects to the SQL Server database using a similar connection string to normal database servers. No operating system maintenance is required by the end-user. The same sizing methodology as the cloud VMs applies to cloud databases.



P1 Premium	P2 Premium	P4 Premium
125 DTUs	250 DTUs	500 DTUs
Up to 500 GB	Up to 500 GB	Up to 500 GB
Geo-Replication	Geo-Replication	Geo-Replication
Point In Time Restore ...	Point In Time Restore ...	Point In Time Restore ...
Auditing	Auditing	Auditing

**Figure 6:** Microsoft Azure SQL DB Sizing Example

Specify DB Details

Instance Specifications

DB Engine

sqlserver-ee

License Model

license-included

DB Engine Version

12.00.5000.0.v1

DB Instance Class

- Select One -

Time Zone (Optional)

- Select One -

Multi-AZ Deployment

db.r3.2xlarge — 8 vCPU, 61 GiB RAM

db.r3.4xlarge — 16 vCPU, 122 GiB RAM

db.r3.8xlarge — 32 vCPU, 244 GiB RAM

Storage Type

- Select One -

Allocated Storage<sup>a</sup>

200 GB

Scaling storage

after launching a DB Instance is currently not supported for SQL Server. You may want to provision storage based on anticipated future storage growth.

**Figure 7:** Amazon RDS SQL Server Instance Sizing Example

Note that storage still needs to be managed for these databases carefully, and that the in-memory demands on storage should be factored into the storage architecture up front when the instance or database is provisioned.

## NETWORKING AND INTERCONNECTS

Every device in the infrastructure stack must communicate with other devices. Server to server communication is handled through Ethernet networking switches and networking adapters on each server. Storage communication can communicate through traditional networking, or can leverage Fibre Channel or InfiniBand technologies for dedicated storage communication channels.

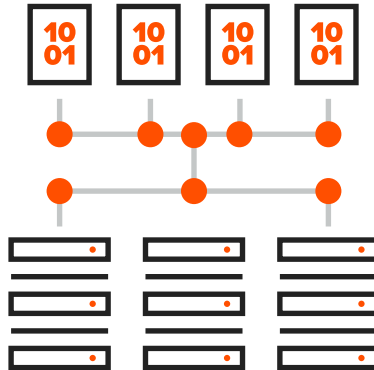
As the storage performance or demand on storage increases, the traffic rates increase. The interconnects between the storage and the servers can become a bottleneck under heavy load, slowing down the operation. Server to server communication can also bottleneck the path in between the servers, slowing communication as well. Special care must be taken to ensure that these communication paths are not overwhelmed during daily operations.

## **UP NEXT**

Now that the infrastructure underneath SQL Server has been reviewed, next up is the first SQL Server in-memory feature – In-Memory OLTP. We will review the architecture and performance characteristics of this flagship performance feature, as well as review the demands on infrastructure and the considerations that must be designed into any critical In-Memory OLTP workload.

## CHAPTER 3

# OPTIMIZING OLTP PERFORMANCE



Microsoft's flagship in-memory technology, In-Memory OLTP, was created out of the desire to improve the performance of the SQL Server engine by reducing the historical assumptions made about memory quantities being smaller than the amount of data to process. The design of the database engine, or at least the design of a new engine to be embedded inside the SQL Server engine, would actively hold all its data within server memory, and only persist changes to disk as necessary.

## CONCEPT

SQL Server's In-Memory OLTP feature is an engine within the engine that allows developers to actively store and manipulate the data in memory, rather than stick with the assumption that all data is on disk like the primary SQL Server engine.

This engine is not the same as simply forcing all the data within a table into memory. Administrators have been trying various tricks of the trade for years to keep all the working data in RAM, but that only works around half of the challenge. The concurrency engine design of the regular OLTP engine still requires the use of locks and latches to manage activity, which maintains the dependency on disk. Two key components make up the new In-Memory OLTP engine. The first is the data containers, the memory-optimized tables and indexes. The second is the optimized means to fetch and manipulate the data, the natively compiled stored procedures.

### ***MEMORY OPTIMIZED TABLES AND INDEXES***

The largest single change between the standard SQL Server storage engine and In-Memory OLTP is that the entire table is stored in memory. Always. You cannot store more data than what you possess in server memory. Because the entire set of data is now always in memory, the concurrency challenges of the traditional on-disk model changed the nature of this new technology. The concurrency model changes and is now truly optimistic. Locking and latching mechanisms are not needed because SQL Server makes changes to data by creating a new row for the change and deleting the original row, rather than updating the row in-place and having to worry about other users accessing the same data at the time of the change. The tables are also now natively compiled into machine code. Every construct necessary to manage and access that table, including metadata and schema, are now optimally stored and accessed by the engine, so that the usage of these tables is the most efficient.

What happens when a user changes data? Is that change important to the business to keep? In-memory tables can be created in two ways. The first is to persist all data changes to disk, or `SCHEMA_AND_DATA` durability. If a process is

deemed transient or unnecessary to persist the changes, such as with temporary operations, the table can be created with `SCHEMA_ONLY` durability. Only the table schema persists between service startups, and any existing data is lost at service shutdown. These tables are beneficial for transient processes that can be restarted or temporary tables that are part of a larger operation. For those operations that must persist, the data change must be written to disk. SQL Server uses the same transaction log structure that the standard engine uses to write these transactions to disk, but writes single log records only when the transaction commits, reducing the strain on the logging process.

### ***IN-MEMORY INDEXES***

What good is storing all the data in memory if you must sift through all of it, sometimes repeatedly, to find the data that the user requests? SQL Server uses indexes in the regular engine to help improve the speed of finding the specific data the user requests. New indexes were created for In-Memory OLTP, and these indexes are also entirely contained within server memory. Two types of indexes are used in In-Memory OLTP – hash indexes and range indexes. A hash index is useful when performing lookups against individual records. A range index is now available for retrieving ranges of values. Traditional index types are not available in in-memory tables.

### ***NATIVELY COMPILED STORED PROCEDURES***

The stored procedures accessing the data in these memory-optimized tables can also be natively compiled into machine code, which reduces the overhead within the query engine layer. Regular stored procedures are compiled when they are interpreted at their first execution time. Calling the natively compiled stored procedure, existing as a DLL, results a significant performance improvement.

### ***CURRENT LIMITATIONS***

The SQL Server In-Memory OLTP engine is effectively an all-new database engine that just happens to be embedded inside the existing engine. As such, it is not feature complete with the primary engine at the time of this writing. The SQL Server 2014 launch feature set shipped with many omissions or partial implementations of key items that many developers use by default. The 2016

release improved its features over the 2014 release, but several limitations still exist that might make adoption of In-Memory OLTP difficult (or impossible). The following partial list of limitations are present in the 2016 release. This list of limitations is improving in each release. For a full list of the current limitations, please visit <https://msdn.microsoft.com/en-us/library/dn246937.aspx>.

- Database integrity checks
- Partitions
- Computed columns
- Replication
- Filestream
- Clustered indexes
- Truncate table
- DROP INDEX
- Common Table Expressions
- Some traditional built-in functions

## SETTING IT UP

Let's set up a sample database for use with In-Memory OLTP. The full set of scripts for this operation are found at <https://github.com/purestorage-partnerconnect/SQLonFlashEBook>. We will create it with a separate filegroup for the In-Memory objects, required by the engine.

```
CREATE DATABASE CMSMem
ON
PRIMARY(NAME = [CMSMem_data],
         FILENAME = 'G:\Data\CMSMem_data.mdf',
         SIZE=500MB),
FILEGROUP [CMSMem_InMem]
CONTAINS MEMORY_OPTIMIZED_DATA
(NAME = [CMSMem_InMem1],
 FILENAME = 'G:\Data\CMSMem_InMem1'),
(NAME = [CMSMem_InMem2],
 FILENAME = 'H:\Data\CMSMem_InMem2')
LOG ON (name = [CMSMem_log],
        Filename='L:\Logs\CMSMem_log.ldf',
        SIZE=500MB);
```

Note the new filegroup that contains the syntax 'CONTAINS MEMORY\_OPTIMIZED\_DATA'. This new file group is specified to contain memory optimized data, and is configured for spanning two data files on two different disks to spread out the workload on multiple disk controllers. Now, set the database to move all isolation levels to snapshot.

```
ALTER DATABASE TestDBInMem SET  
MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT = ON;
```

For comparison's sake, we will also create a logically similar database called CMS with no In-Memory OLTP active in the database. It will contain the same logical schema and data, and we will test certain operations between these two databases. The scripts for creating this database are also found at <https://github.com/purestorage-partnerconnect/SQLonFlashEBook>. To demonstrate the power of this feature, I want a real-world example rather than a contrived example. Let's create a sample scenario to develop against. The scenario is querying for people living in different states in the USA. First, create a table to store their information.

```
CREATE TABLE dbo.People (  
    [ID] [int] IDENTITY(1,1) NOT NULL ,  
    [Name] varchar(32) NOT NULL  
        INDEX IX_People_Name HASH  
        WITH (BUCKET_COUNT = 75000000),  
    [City] varchar(32) NOT NULL  
        INDEX IX_People_City HASH  
        WITH (BUCKET_COUNT = 50000),  
    [State_Province] varchar(32) NOT NULL  
        INDEX IX_PEOPLE_State_Province HASH  
        WITH (BUCKET_COUNT = 1000),  
    PRIMARY KEY NONCLUSTERED HASH (ID) WITH  
        (BUCKET_COUNT = 75000000)  
) WITH (  
    MEMORY_OPTIMIZED = ON,  
    DURABILITY = SCHEMA_AND_DATA);
```

Note the syntax changes on the table creation command. We are creating a hash index on some of the columns. A hash index is an array-based index of a certain number of slots, with each slot pointing to the memory address of a particular row in the table. I also want this data to persist between reboots, so the durability parameter is set to 'SCHEMA\_AND\_DATA' instead of just 'SCHEMA'.

One major point of misunderstanding is with the BUCKET\_COUNT value. The bucket count roughly represents the number index positions in the index. The hashing function used to identify the unique slots is good but not perfect, and hash collisions will happen, which means that two different keys can hash to the same bucket value. In-Memory OLTP solves this by chaining the two with updated index pointers. Thus, most tables should use a bucket count value of roughly 1.5 to two times the number of unique items that you expect to be loaded into this table. I want to load a significant number of fake users into this table, so I set the bucket count to 75M. I only have a small number of cities, states, and countries in the geographical tables in the database, and as such, the bucket counts for these items will be much smaller. Keep in mind that this setting directly relates to the size of the hash index. The calculation for this value is:

$$[Hash\ index\ GB] = \frac{(8 * [Actual\ Bucket\ Count])}{1073741824\ (Bytes\ in\ 1\ GB)}$$

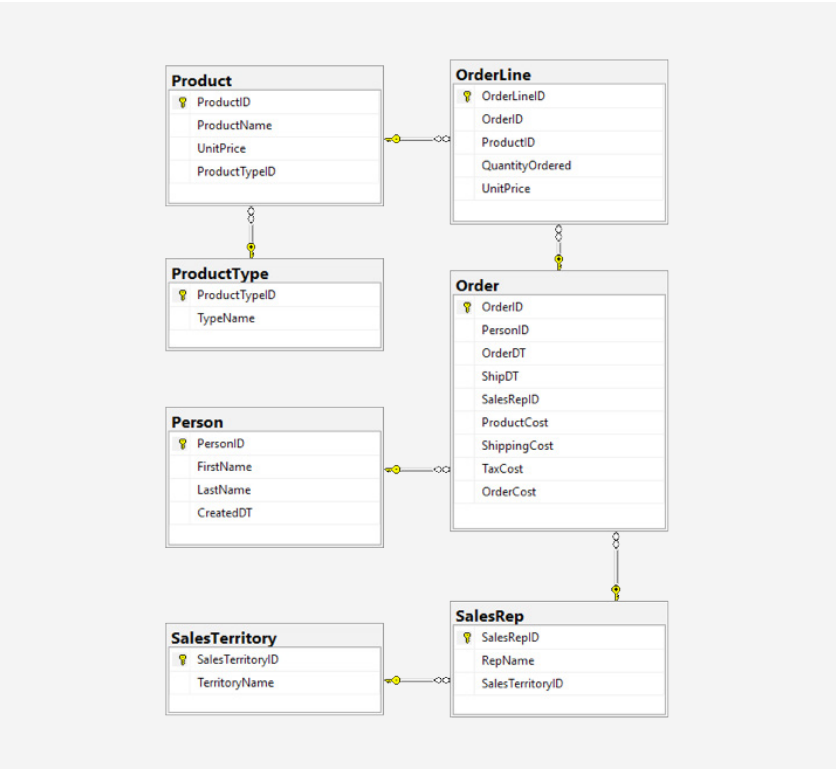
To identify the number of buckets of the in-memory indexes, execute the following query.

```
SELECT
    OBJECT_NAME(s.object_id) AS TableName,
    i.name as IndexName,
    s.total_bucket_count
FROM
    sys.dm_db_xtp_hash_index_stats s
    INNER JOIN sys.hash_indexes i
        ON s.object_id = i.object_id
        and s.index_id = i.index_id
ORDER BY
    TableName, IndexName
```



**CMS DATABASE**

The remainder of the CMS database is straightforward in its nature.



Orders are placed using line items consisting of products, each with their own product type. People are the recipients of the orders. Sales representatives assist with and enter in the order details, and each sales rep has a sales territory (in this case United States states).

To load the data into the database, a stored procedure called `dbo.Order_Generate` was created to load randomly generated data into the CMS database schema. Once loaded into the on-disk database, staging tables were used to copy the data into the In-Memory OLTP database. This process of loading and cloning the data takes some time, so backup each database once this operation is complete for future database re-use.

The sample testing was performed with 12,000 products, 5.2M orders, 265M order line items, and 5,000 sales representatives. All test operations performed for this document were performed on a SQL Server 2016 SP1 Enterprise Edition virtual machine with 32 vCPUs and 128GB RAM.

## POTENTIAL PERFORMANCE IMPROVEMENT

First and foremost, simply using In-Memory OLTP is not guaranteed to improve the performance of your application. In some cases, it can even substantially hinder your performance. However, when used correctly in the right circumstance, the performance improvements are impressive. For example, querying these tables for product sold count and value for a given month produces a noticeable improvement in performance.

The stored procedure `dbo.Product_By_Date` was created containing a query to produce this sales by date report. The on-disk database contains the following T-SQL.

```
USE [CMS]
GO

CREATE proc [dbo].[Product_by_Date] (
    @StartDT DATETIME,
    @EndDT DATETIME
) AS

SET NOCOUNT ON;

SELECT
    I.ProductID,
    SUM(I.QuantityOrdered) AS SoldCount,
    SUM(I.UnitPrice) AS SoldValue
FROM
    dbo.[Order] o
    INNER JOIN dbo.OrderLine I
```

```

        ON o.OrderID = l.OrderID
    INNER JOIN dbo.Person p
        ON p.PersonID = o.PersonID
    INNER JOIN dbo.SalesRep r
        ON r.SalesRepID = o.SalesRepID
    INNER JOIN dbo.SalesTerritory t
        ON t.SalesTerritoryID = r.SalesTerritoryID
WHERE
    o.OrderDT >= @StartDT
    AND o.OrderDT <= @EndDT
GROUP BY
    l.ProductID
ORDER BY
    SoldValue DESC

```

The equivalent stored procedure, constructed for In-Memory OLTP, utilized natively compiled stored procedures to boost the performance of the operation.

```

USE [CMSMem]
GO

CREATE OR ALTER PROC [dbo].[Product_by_Date] (
    @StartDT DATETIME,
    @EndDT DATETIME
)
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS

BEGIN ATOMIC WITH (
    TRANSACTION ISOLATION LEVEL = SNAPSHOT,
    LANGUAGE = N'us_english')

SELECT
    l.ProductID,
    SUM(l.QuantityOrdered) AS SoldCount,
    SUM(l.UnitPrice) AS SoldValue

```

```

FROM
    dbo.[Order] o
    INNER JOIN dbo.OrderLine l
        ON o.OrderID = l.OrderID
    INNER JOIN dbo.Person p
        ON p.PersonID = o.PersonID
    INNER JOIN dbo.SalesRep r
        ON r.SalesRepID = o.SalesRepID
    INNER JOIN dbo.SalesTerritory t
        ON t.SalesTerritoryID = r.SalesTerritoryID

WHERE
    o.OrderDT >= @StartDT
    AND o.OrderDT <= @EndDT

GROUP BY
    l.ProductID

ORDER BY
    SoldValue DESC

END;

```

Note that the actual query is identical to the on-disk database. The natively compiled stored procedure requires a few additional commands to successfully compile, such as `SCHEMABINDING` because of the nature of compiled objects, and `BEGIN_ATOMIC` because the command must contain only one operations block.

Executing this stored procedure repeatedly against the on-disk database with the parameters for June 2016 resulted in an average runtime of 24 seconds.

```

USE CMS
GO
EXEC dbo.Product_By_Date '2016-06-01', '2016-06-30'
GO

```

Executing this same stored command against the in-memory database yielded improved results.

```
USE CMSMem
GO
EXEC dbo.Product_By_Date '2016-06-01', '2016-06-30'
GO
```

The performance differences are impressive. The in-memory table, coupled with the natively compiled stored procedure, executed the same query in just three seconds, a nine times improvement.

Additional improvements are seen with many queries against these databases. The same time span was used with these stored procedure calls. As you can see, not every scenario resulted in large performance gains, but for those that improved, the results were substantial.

STORED PROC	ON DISK RUNTIME (SEC)	IN-MEMORY RUNTIME (SEC)
Product_by_Date	24	3
Produc Product_by_Territory_Date	12	12
Product_Ship_Delays_by_Age_Date	0.353	0.363
Product_Ship_Delays_by_Date	118	0.860
Sales_By_Territory_Rep_Date	0.324	0.282

Additional performance is also exhibited when performing large updates. The majority of the performance improvements are shown to be when the operations were dependent on the logging to disk. For example, the act of reassigning sales reps can be expensive when on disk. Thousands of orders need to be updated, and each operation is time consuming. Performing this same operation on in-memory objects is noticeably less expensive.

```

--Update random sales reps
SET STATISTICS TIME ON
GO
USE CMS
GO
DECLARE @OldRepID INT, @NewRepID INT
SELECT TOP 1 @OldRepID = SalesRepID FROM dbo.SalesRep ORDER BY
NEWID()
SELECT TOP 1 @NewRepID = SalesRepID FROM dbo.SalesRep ORDER BY
NEWID()
-- So we can transpose the item to the
-- in-memory command for continuity in operations
PRINT @OldRepID
PRINT @NewRepID

EXEC dbo.Sales_Rep_Reassign @OldRepID, @NewRepID
GO
--148ms

USE CMSMem
GO
EXEC dbo.Sales_Rep_Reassign 4652, 1265
GO
--6ms

```

The time savings in this scenario is tremendous. The runtime improves from 148ms to just 6ms. As with any new feature within any application, your experiences with the performance gains (or decreases) will vary on your situation. Used wisely, In-Memory OLTP can improve the performance of certain operations by a considerable margin.

## STORAGE CONSIDERATIONS

The storage considerations when using In-Memory OLTP are substantial. If the table is configured to be non-durable, meaning that the data never persists to disk and is lost if the service or server is restarted, has little demands on storage. However, more frequently are use cases where the data must persist to disk, and

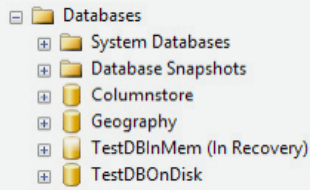
any change to data must be written to disk before the operation can complete. Note the durability parameter when creating this in-memory table.

```
CREATE TABLE dbo.People (  
    [ID] [int] IDENTITY(1,1) NOT NULL ,  
    [Name] varchar(32) NOT NULL  
        INDEX IX_People_Name HASH  
        WITH (BUCKET_COUNT = 60000000),  
    [City] varchar(32) NOT NULL  
        INDEX IX_People_City HASH  
        WITH (BUCKET_COUNT = 60000000),  
    [State_Province] varchar(32) NOT NULL  
        PRIMARY KEY NONCLUSTERED HASH (ID) WITH  
        (BUCKET_COUNT = 60000000)  
) WITH (  
    MEMORY_OPTIMIZED = ON,  
    DURABILITY = SCHEMA_AND_DATA);
```

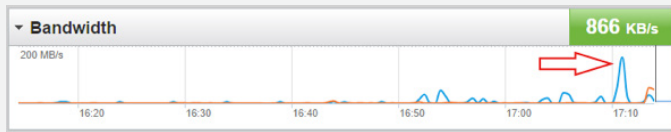
The table was created with a durability setting for making both the table definition and the contents of the table persist to disk. How fast can your storage write these changes to disk? Your in-memory database is now limited to the performance of your disk's ability to write the log buffers to disk.

When considering leveraging In-Memory OLTP in your applications, perform a close examination of your storage's ability to write small blocks of data to disk very quickly. Transactional latency in these operations will slow down any in-memory change operation.

Additionally, the high-performance demands on storage to read all of the data while SQL Server starts up and rebuilds all of the in-memory indexes is enough to cause a significant drain on storage performance resources during this operation, and will cause the SQL Server database to be delayed in starting up.



**Figure 8:** In-Memory OLTP Database Startup Delays



**Figure 9:** Burst Startup I/O Consumption

All-flash storage possesses the high-performance characteristics to be able to accommodate this level of performance with a minimal amount of overhead to the data change operation. Other more traditional types of storage do not contain the necessary equipment needed to sustain these performance levels under time.

## CPU CONSIDERATIONS

No feature to boost performance of key operations is free. As the bottlenecks shift away from the historically primary bottleneck of storage, the bottlenecks shift upwards in the system stack. If the data is stored in memory and disk-bound operations are no longer the overarching bottleneck to performance, chances are that you will see your CPU consumption elevate as the workload shifts. Sometimes this CPU consumption is substantial, and you do not want to experience CPU pressure during critical time periods.



As you explore In-Memory OLTP, I strongly recommend continuous CPU consumption metric collection. Quite a few great third-party applications are on the market to help you collect this information, or you can use the free Windows Server Perfmon utility that comes included with every modern version of Windows. Understand the before and after CPU consumption metrics as you test and adopt this feature. Adding more CPUs are trivial in a virtual machine or cloud-based database (license discussion excluded of course), and understanding the new bottleneck as the system adjusts to your change of utilization.

## **MEMORY CONSIDERATIONS**

As the bottleneck shifts away from primary storage and all the memory is leveraged for data storage, the amount and performance of the memory is more important than ever. Above all else, to use In-Memory OLTP for your workloads, you must have more memory than the data you are attempting to store in memory. Beyond the obvious, performance challenges exist with in-memory operations as well. Many factors contribute to the performance of the memory on your server, including:

- VM memory hot-add enabled (some hypervisors disable in-guest vNUMA presentation when this feature is enabled)
- Virtual machine NUMA imbalance
- Virtual machine memory pressure from the hypervisor
- Misconfigured physical server memory configuration (triple channel memory disabled, or slower than expected memory speed due to suboptimal memory placement on the system mainboard)

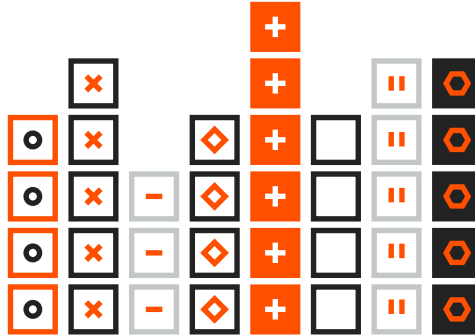
Work with your system administrators to ensure that your server is configured for the maximum possible memory performance.

## UP NEXT

Next up is a deep-dive on how SQL Server Columnstore indexes are configured and used, demonstrations on the potential improvements from using them, and the impact on the infrastructure underneath the SQL Server that the administrators need to monitor and architect for.

## CHAPTER 4

# CREATING EFFICIENCY WITH COLUMNSTORE



Traditional SQL Server indexes (sometimes known as a RowStore) group and store database table data for each row and then group the rows together to form an index. For some types of OLTP workloads, where the index necessary to boost performance is not entirely predictable, these indexes have worked well for years. However, certain types of workload, such as larger aggregate reporting and warehouses, can benefit from a different indexing strategy. Columnstore indexes are designed for use with large tables.

Columnstore indexes are essentially a pivot of a traditional index. Rowstore indexes (the normal indexes you are used to) are stored row by row.

ID	PersonID	OrderDate	OrderAmount
1	10964	01/24/2016 18:29:42	\$5,678.48
2	11674	02/06/2016 04:53:27	\$4,842.68
3	12479	03/25/2016 04:53:14	\$8,226.31
4	13484	04/16/2016 13:28:34	\$7,974.89

*Traditional Rowstore Indexes*

In Columnstore indexes, data is grouped by column instead of row, and is stored one column at a time, then grouped by the rows to form an index.

ID	PersonID	OrderDate	OrderAmount
1	10964	01/24/2016 18:29:42	\$5,678.48
2	11674	02/06/2016 04:53:27	\$4,842.68
3	12479	03/25/2016 04:53:14	\$8,226.31
4	13484	04/16/2016 13:28:34	\$7,974.89

*Columnstore Index*

Re-orienting the index per column is more in line with the nature of some workloads, and the improved efficiency of such an index can lead to a significant performance improvement in query runtime. Workloads such as decision support systems, roll-up reporting, and warehouse fact tables can benefit from Columnstore indexes. Microsoft claims that these workloads can experience up to a ten times performance improvement, all while saving up to seven times disk space because of the high levels of compression versus normal indexes.

## CONCEPT

With the release of SQL Server 2012, Microsoft introduced non-clustered columnstore indexes into the core SQL Server engine. Based on the VertiPaq storage engine's in-memory data compression technology, later renamed xVelocity in-memory analytics engine, columnstore indexes store indexes by column instead of per row. Columnstore indexes use a columnar formatted index structure instead of the traditional B-tree format of row-wise clustered and nonclustered traditional indexes. Each column is stored in separate data pages, so only the data necessary for fulfilling a specific query is accessed.

Each column in the columnstore index is its own segment, and can only store values from that column. A segment is a compressed Large Object (LOB) that is optimized for up to one million rows. Multiple segments can make up a columnstore index. Segments are loaded into memory upon being queried instead of the traditional page or extent.

The xVelocity engine is also optimized for parallel data scanning, and can utilize the multi-core processors in modern server hardware. The data stored in-memory is also significantly compressed, so more data can be contained in memory and less is to be read from disk.

Originally released only for nonclustered indexes, subsequent releases have introduced support for clustered indexes as well. As of SQL Server 2016 SP1, columnstore indexes are now available on Standard Edition. Columnstore indexes are also available in premium Azure databases.

## CONFIGURATION AND USAGE

Columnstore indexes come in two flavors (as of SQL Server 2016): clustered and nonclustered. Creating one is as simple as creating a traditional index. For example, let's create a clustered columnstore index on a simple table.

	COLUMN NAME	DATA TYPE	ALLOW NULLS
▶	PerfDataID	bigint	<input type="checkbox"/>
	ServerID	int	<input type="checkbox"/>
	BatchLoadID	int	<input checked="" type="checkbox"/>
	PerfmonCounterInstanc...	int	<input checked="" type="checkbox"/>
	PerfValue	float	<input checked="" type="checkbox"/>
	CollectionDT	datetimeoffset(2)	<input checked="" type="checkbox"/>

We do not have a primary key defined on this table at this time, but the column PerfDataID is defined as a bigint IDENTITY column. The syntax to create the clustered columnstore index is as follows.

```
CREATE CLUSTERED COLUMNSTORE INDEX [cci_PerfData] ON
[dbo].[PerfData]
GO
```

That's it! Nonclustered indexes are very similar. For the same table, just structured with a traditional rowstore index, let's create a nonclustered clustered index on a few of the columns used for reporting.

	COLUMN NAME	DATA TYPE	ALLOW NULLS
🔍	PerfDataID	bigint	<input type="checkbox"/>
	ServerID	int	<input type="checkbox"/>
	BatchLoadID	int	<input checked="" type="checkbox"/>
	PerfmonCounterInstanc...	int	<input checked="" type="checkbox"/>
	PerfValue	float	<input checked="" type="checkbox"/>
	CollectionDT	datetimeoffset(2)	<input checked="" type="checkbox"/>

```

CREATE NONCLUSTERED COLUMNSTORE INDEX IX_PerfData_
NCCICover01 ON dbo.PerfData
(
    ServerID, PerfmonCounterInstance, PerfValue
)
GO

```

That's it as well! Microsoft presents an incredible number of options for use with columnstore indexes, and reading the MSDN page at <https://msdn.microsoft.com/en-us/library/gg492153.aspx> would be good before you start creating these in production. The space consumed by these indexes are quite impressive in their differences. Comparing the three scenarios – rowstore indexes, clustered columnstore indexes, and rowstore primary key indexes and a nonclustered columnstore index – shows the significant disk utilization differences. This query will show us the space consumed by the table and indexes. The results are impressive.

```

SELECT
    s.Name AS SchemaName,
    t.NAME AS TableName,
    p.rows AS RowCounter,
    SUM(a.total_pages) * 8 / 1024 AS TotalSpaceMB
FROM
    sys.tables t
INNER JOIN
    sys.indexes i ON t.OBJECT_ID = i.object_id
INNER JOIN
    sys.partitions p ON i.object_id = p.OBJECT_ID AND i.index_id =
    p.index_id
INNER JOIN
    sys.allocation_units a ON p.partition_id = a.container_id
LEFT OUTER JOIN
    sys.schemas s ON t.schema_id = s.schema_id
WHERE

```

```

        t.NAME = 'PerfData'
GROUP BY
        t.Name, s.Name, p.Rows
ORDER BY
        t.Name
GO

```

TYPE	ROWCOUNT	MB
Rowstore	120,436,320	11,104
Clustered Columnstore	120,436,320	714
Rowstore PK and NC Columnstore	120,436,320	7460

The performance differences on the aforementioned reporting and warehousing queries are quite pronounced as well. A simple aggregate reporting query on this table is as follows.

```

SELECT
    MIN(PerfValue) AS MinVal,
    AVG(PerfValue) AS AvgVal,
    MAX(PerfValue) AS MaxVal
FROM
    dbo.PerfData d
    INNER JOIN dbo.PerfmonCounterInstance i ON
        i.PerfmonCounterInstanceID =
        d.PerfmonCounterInstanceID
    INNER JOIN dbo.PerfmonCounter c ON
        c.PerfmonCounterID = i.PerfmonCounterID
    INNER JOIN dbo.PerfmonCounterSet s ON
        s.PerfmonCounterSetID = c.PerfmonCounterSetID
WHERE
    d.ServerID = 8
    AND s.PerfmonCounterSetID = 23
GO

```



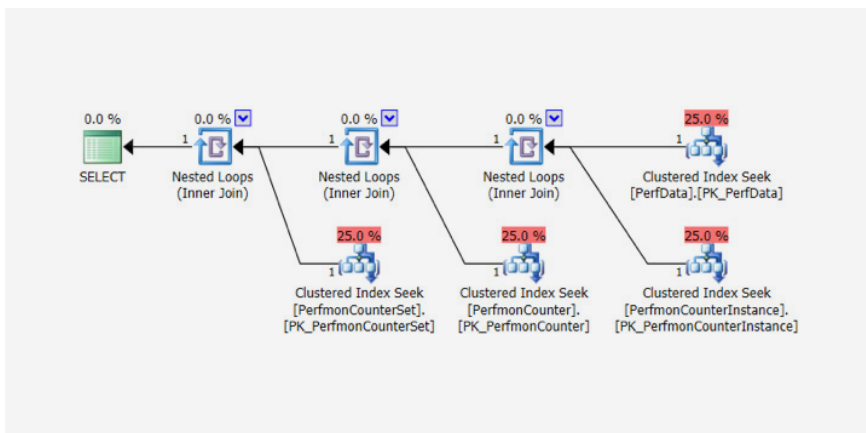


However, when performing a single value lookup instead of an aggregate query, the performance characteristics change. The runtimes are very low because the working set is in memory, but the differences are present.

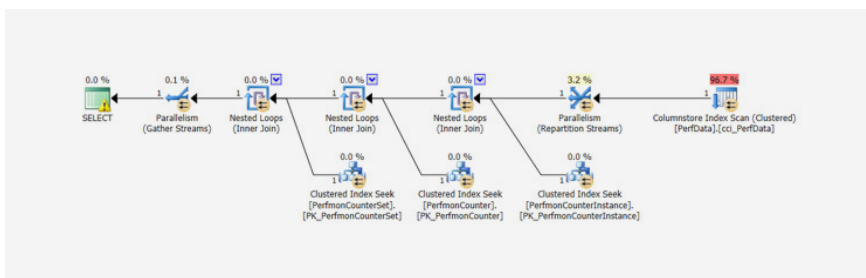
```
SELECT
    d.PerfDataID, d.ServerID, d.PerfValue, d.CollectionDT
FROM
    dbo.PerfData d
    INNER JOIN dbo.PerfmonCounterInstance i ON
        i.PerfmonCounterInstanceID =
        d.PerfmonCounterInstanceID
    INNER JOIN dbo.PerfmonCounter c ON
        c.PerfmonCounterID = i.PerfmonCounterID
    INNER JOIN dbo.PerfmonCounterSet s ON
        s.PerfmonCounterSetID = c.PerfmonCounterSetID
WHERE
    d.PerfDataID = 456789
GO
```

TYPE	RUNTIME (MS)
Rowstore	5
Clustered Columnstore	23
Rowstore PK and NC Columnstore	4

As you can see, querying the table to use the columnstore index requires nothing new in your query. Microsoft makes the use of this feature transparent to the actual query syntax itself. However, the execution plan varies between index types. For the traditional rowstore index, the execution plan is as follows.



The nonclustered columnstore index query's execution plan is identical to this execution plan since the lookup value was off of the primary key column, which is not in the clustered index. The clustered columnstore index execution plan is very different.



SQL Server also complained of missing an index on the ID column, and suggested that the developer place a nonclustered index on that column to improve performance. Test accordingly as you deploy columnstore indexes to ensure that your workload performance responds in a positive manner before you deploy the new indexes to production.

## STORAGE IMPLICATIONS

While the storage footprint of storing columnstore indexes is much lower than with traditional indexes, the speed of the storage is equally as important as with In-Memory OLTP. Lower latency to disk means that the fetching of the columnstore

index segments from disk are improved, and the data is loaded into memory for query processing faster. Unlike In-Memory OLTP, which requires all the data to be resident in memory at all times, columnstore indexes are persisted on disk like normal index. They can be pushed out of the buffer pool, and will need to be re-fetched as needed. Lower latency and faster throughput will help the data be loaded back into memory faster, which improves the performance of the columnstore index.

## LIMITATIONS

Currently columnstore indexes do have some limitations, but SQL Server 2016 has removed the majority of the challenges with using columnstore indexes. One example that persists is that nonclustered columnstore indexes cannot be created on a table with an existing clustered columnstore index. However, most of the previous limitations have been lifted.

The functionality and use of these indexes have changed since its introduction in SQL Server 2012, and previous versions of SQL Server have many more limitations in usage, such as read-only columnstore indexes. For a detailed list of the differences, please visit <https://msdn.microsoft.com/en-us/library/dn934994.aspx> for more details.

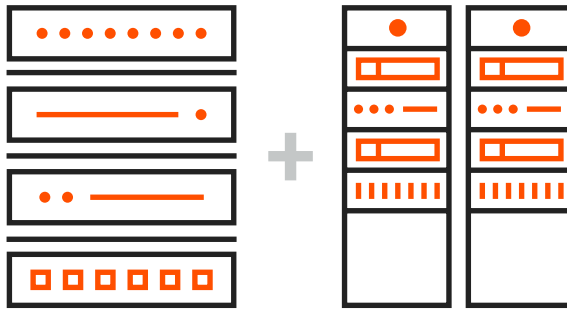
Now, columnstore indexes are not always the right solution for all workloads. Columnstore indexes are geared for warehousing and DSS-type workloads, and are more appropriately designed for improving the performance of scan-type operations. Seek operations, such as individual row lookups, could perform worse than a properly designed traditional rowstore index strategy. However, used appropriately, columnstore indexes are an incredibly powerful feature, and thanks to the addition of columnstore indexes in SQL Server 2016 SP1, can come in handy quite frequently.

## UP NEXT

In-memory functions in SQL Server are not mutually exclusive. In-Memory OLTP and Columnstore indexes can be used together to provide dramatically improved performance for your varied workloads.

## CHAPTER 5

# AN INTEGRATED APPROACH TO MANAGING DATA PERFORMANCE



Microsoft has coined a new phrase with the release of SQL Server 2016, calling the act of performing analytics from an OLTP system “Real-Time Operational Analytics”. Historically, businesses have needed to separate their OLTP workloads from their reporting environments and have complex means to synchronize the data to keep the division of OLTP and reporting from negatively impacting both. SQL Server 2016 allows users to perform reporting, warehousing, and ETL workloads off their OLTP environments by allowing the use of both In-Memory OLTP and columnstore indexes at the same time.

By leveraging these new features, the two traditionally distinct environments are simplified by removing the secondary reporting system. Historically, the systems were set up very differently, and the two configurations could not function well together. OLTP workloads are very random and accommodate highly concurrent workloads of small transactions.

Warehousing-type workloads are larger in scale, and usually perform aggregate functions on larger sets of data. The database instance configuration, database setup, and indexing strategy are usually very different per role. Setting up an OLTP environment to handle reporting tasks meant compromise on performance for both workload types. The benefits of combining these roles without compromising performance and complexity are significant. Space is saved on the SAN. The additional system no longer needs to be managed and secured. Licensing could be reduced. But, the largest improvement is that the reporting environment no longer lags behind the OLTP system. End-users can now produce real-time reports with none of the traditional delay that accompanied the data synchronization.

## USAGE

Microsoft could not have made using these two features together simpler. The table structure is now very similar to a traditional table, except that the index creation is performed in-line with table creation. One difference in usage from traditional tables is that the columnstore index must include all the columns in the in-memory table. As an example, the `dbo.Order` table from Chapter 3 was modified for use with clustered columnstore indexes. That's it!

```
CREATE TABLE [dbo].[Order]
(
    [OrderID] [bigint] IDENTITY(1,1) NOT NULL
        PRIMARY KEY NONCLUSTERED,
    [PersonID] [bigint] NOT NULL,
    [OrderDT] [datetime] NOT NULL,
    [ShipDT] [datetime] NULL,
    [SalesRepID] [int] NOT NULL,
    [ProductCost] [numeric](18, 2) NULL,
    [ShippingCost] [numeric](10, 2) NULL,
    [TaxCost] [numeric](10, 2) NULL,
    [OrderCost] [numeric](18, 2) NULL,
    INDEX PK_Order_CCI CLUSTERED COLUMNSTORE
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
GO
```

## WRAP UP

In-memory functions in SQL Server are another fantastic tool in the SQL Server toolbox. When used for the right use cases, these new features in SQL Server are incredibly powerful. SQL Server In-Memory OLTP is a great means to improve the speed of certain operations by moving the dataset entirely to memory with the lock and latch-free architecture. Columnstore indexes can help add warehousing and decision-support roles to existing OLTP servers in a way that will not negatively impact the OLTP server itself. Leveraging both at the same time will help you achieve what Microsoft calls “Real-Time Operational Analytics”, all with minimal to no changes in your current codebase.

The only challenge is to ensure that your infrastructure underneath these in-memory features is up to the task of handling the shift in the resource demand. The memory speed must be as fast as possible. The storage performance is vital to the overall performance of the in-memory functions, as any changes to this data must be persisted to disk. Enterprise storage that exhibits very low latency and high throughput is critical to ensuring that your in-memory feature usage does not backfire and perform worse than without using these features.

As with any other new toy, test your code as you explore these new features. Understand the impact of these on the environment around your databases, such as the unwritten dependency on fast, low-latency storage or the requirement for appropriate amounts of memory, and plan accordingly before you deploy these features to production.

Used properly, these two exciting features can help your organization improve how it uses data, and can help accelerate your business ahead of your competition!





**SEE IT IN ACTION - SCHEDULE A DEMO AT:**

<http://www.purestorage.com/forms/schedule-a-demo.html>



**LEARN MORE ABOUT ALL-FLASH STORAGE  
OPTIONS FOR SQL SERVER AT:**

<https://www.purestorage.com/solutions/applications/microsoft.html>



© 2017 Pure Storage, Inc. All rights reserved. Pure Storage and the "P" logo are trademarks or registered trademarks of Pure Storage in the U.S. and other countries. All other trademarks are the property of their respective owners.