Use Case Comparison of ASE 15.7 on Solid State Disks [Part 1]

By Mich Talebzadeh

In this article, we will consider the use case of Solid State Disks for Sybase ASE database devices and the advantage these devices offer compared to the traditional Hard Disk Drives. We will consider ASE's read intensive activities in part one of the article and the write intensive activities in part two.

Mich Talebzadeh is an award winning consultant and a technical architect who has worked with the database management systems since his student days at Imperial College, University of London, where he obtained his PhD in Experimental Particle Physics. He specializes in the strategic use of Sybase and Oracle. Mich is the author of the book "A Practitioner's Guide to Upgrading to Sybase ASE 15", the co-author of "Sybase Transact SQL Programming Guidelines and Best Practices" and the author of the forthcoming books "Complex Event Processing in Heterogeneous Environments", "Oracle and Sybase, Concepts and Contrasts" and numerous articles. Mich can be reached at mich@peridale.co.uk.

Database performance is ultimately constrained by the need to read and write data from a persistent storage source. Indeed this is the primary purpose of the database. For almost the entire history of ASE and other databases, this persistent storage has been provided by the magnetic spinning hard disks (HDD). The relative performance of magnetic disk has been so poor compared to the rest of the hardware, that database performance tuning has focused primarily on minimizing the disk IO. Only now, with the advent of Solid State Disk (SSD) technology and in-memory databases (IMDB), we are getting to the point of an increase in IO performance to match the improvements we have come to take for granted in CPU and memory access speeds.

Although replacing all the storage components of ASE with SSD is a rather expensive option for many shops, ASE servers can benefit from an intelligent use of Solid State Disk. However, it is not always obvious how best to leverage limited quantities of SSD in order to get the best performance return from the investment. In this paper and part two, we will look at the areas where SSD technology will contribute to the performance of ASE. Making sure that the database will take full advantage of solid state storage is the next big step in solid state adoption.

Why SSD

The fundamental technology of the traditional magnetic disk has remained the same since its inception. One or more platters contain magnetic charges that represent bits of information. These magnetic charges are read and written by an actuator arm, which moves across the disk to a specific position on the radius of the platter, and then waits for the platter to rotate to the appropriate location. The time taken to read an item of data is the sum of the time taken to move the head into position – *seek time*, the time taken to rotate the item into place – *rotational latency* and the time taken to transmit the item through the disk controller – *transfer time*. The whole of this process creates lag time and reduces the device's lifetime. In contrast, SSD having only electronic components are much better adapted by design in both performance and capacity. The two main categories of Solid State Disk technology are:

- DDR RAM-based SSD, in which memory not very different from computer RAM is used as the storage medium. A capacitor or battery is incorporated so that data can be written to a non-volatile storage medium in the event of a power failure.

- Flash-based SSDs, in which data is stored in NAND flash memory similar to that used in USB drives. This memory is non-volatile, so it will not lose data in the event of a power failure. For the purpose of our tests, we deployed flash-based SSD.

Flash-based Solid State Disks have a three-level hierarchy of storage. Individual bits of information are stored in cells. In a single-level cell (SLC) SSD, each cell stores only a single bit. In a multi-level cell (MLC), each cell may store two bits of information. MLC SSD devices consequently have greater storage densities, but lower performance and reliability. Cells are arranged in pages typically 4K in size and pages into blocks of typically 512K. In summary, SSD avoids the key negative aspects of the conventional HDD. SSD does not have to spin, thus writing to any area on the device is just as fast. So the argument is that SSD is faster, but how fast and where.

Where to Use SSDs with ASE

An RDBMS is charged with three basic tasks. It must be able to put data in, keep that data, and take the data out and work with it. So with that in mind, it comes to where best to apply this solid state technology. There is an abundance of argument that improving the speed of ASE devices for tempdbs and the system databases such as master and sybsystemprocs is paramount to better performance and thus SSD is best applied to these areas. Well that is not surprising as most of these areas are on the critical path to better performance. To understand the case better, we will need to understand the conditions under which SSD will be most valuable dealing with *user databases*. Thus, a frequently asked question is how to improve the response of a given production database under different loads? This is especially true for ASE databases that tend to be OLTP in nature with considerable random IO's via index search. In contrast, we will show that SSD is not going to be that much faster for table scans. This is because SSD scan rates are throttled by transfer rates, just as the magnetic disk is. When the magnetic disk does not have to spin either because it is writing all the data to the same location, SSD is not much faster.

Environment Set-up

This is one of those series of experiments that you can do for yourself. I did the whole of this test on my server. My server has I7-980 HEX core processor with 24GB of RAM and 1 TB of HDD SATA II for test/scratch backup and archive. The operating system was RHES 5.2 64-bit installed on a 120GB OCZ Vertex 3 Series SATA III 2.5-inch Solid State Drive.

While your setup will almost certainly be different to mine you should still see similar results. However, if they are measurably different, a detailed examination of the configuration differences would be in order to establish the reasons for differences that cannot be covered in a short article like this one. Worth noting is that these tests were done is isolation when the only application running on ASE was the test itself. Some will argue that these outcomes do not apply to production environment with concurrency and multiple users. I counter argue that given that performance is a deployment issue associated with production, then through experience one can argue that nothing comes that close to emulating a production environment. Additionally, no testing done in production environment can be considered a safe test in isolation. There are often myriad of activities in production environments that do skew any test results. These may not be correlated to ASE itself.

ASE version installed was ASE Developer's Edition 15.7 64-bit, EBF 19688 SMP ONE-OFF/P/x86_64. The binaries were created on HDD. ASE itself was configured with 18GB of *max memory*, of which 1GB was allocated to *procedure cache*, 8GB to *default data cache*, 5GB to *in-memory database cache*, 640MB to *log cache*, 640MB to *tempdb data cache* and 400MB to *tempdb log cache*.

For ASE system databases including tempdbs plus user databases (excluding databases purposely created on HDD and the in-memory database, see later), I used file systems on 240GB OCZ Vertex 3 Series SATA III 2.5-inch Solid State Drive. With 4K Random Read at 53,500 IOPS and 4K Random Write at 56,000 IOPS (manufacturer's figures), this drive is one of the fastest commodity SSDs using NAND flash memory with Multi-Level Cell (MLC) and has the advantage of very low latency. In summary, the hardware I used was more than adequate for my needs. Note that in my server all drives were mounted internally. All devices for user databases were created with *directio = true*.

Access to the server from local PCs was provided via Ethernet network setup using open source terminal emulator Putty and I used WINSCP416 for file transfer between PCs and Servers.

Approach

We will follow the established method of forming a hypothesis for which an outcome can be deduced. Following that, we will develop a test case that proves or otherwise that deduction.

Scenario 1: SSD Versus HDD for Random IOs via an Index Scan

Expected Outcome

SSD would show its use for random IO's done via an index as seek time disappears. In short, if we want to measure what this does for me IO wise, we just measure the reads and do it as simply as possible.

In Summary

- Typically, fast random access for reading, as there is no read/write head to move.
- Extremely low read latency times, as SSD seek times are orders of magnitude lower than the best current hard disk drives.

The result should show significant gain for SSD when compared with HDD and the seek time as there is no read/write head to move.

So to prepare a test case we want to read data randomly from the disk. We would like to read every row of the table by index – one for every single row we retrieve from the underlying table in a format index –> table, index –> table, something that happens daily in ASE.

ASE Wait Times

Waits times in ASE are calculated from MDA tables and we will need some input for our tests. While it can be done loosely (within certain orders of magnitude), much like other databases it is not precise and it is more suspect to misses and rounding for shorter run times, much like any other engine. The reason is that different counters are maintained in different structures and as a result get incremented at different points in time based on when/where the counter is in the codepath. To make matters complicated, some counter structures use the timeslice while others use the clock ticks and try to convert that to time.....so, for example, monProcessWaits and monSysWaits use timeslices (default of 100ms) while monSysStatement and monProcessActivity looks at the number of clock ticks. The result that you see with everything in on monProcessWaits or monSysWaits with WaitTime ending in even 100ms increments is the result. The issue is that if a wait is < 1 timeslice, it reports a 0 WaitTime effectively – so the WaitTime can be skewed a bit low. CPUTime is always computed as ElapsedTime – WaitTime.

In conclusion we default a Wait to 1ms (vs. 0). We believe this is likely a good choice that may or may not be accurate as ASE can context switch faster than 1ms....but is probably more accurate than 0ms –> 100ms increments.

Test Case

We start with a table of random data. We will call this table t and we ensure that this table has enough rows to provide meaningful test values. The structure of this table is as follows:

```
CREATE TABLE t
(
 OWNER                varchar(30)      NOT NULL,
 OBJECT_NAME          varchar(30)      NOT NULL,
 SUBOBJECT_NAME       varchar(30)      NULL,
 OBJECT_ID            bigint           NOT NULL,
 DATA_OBJECT_ID       bigint           NULL,
 OBJECT_TYPE          varchar(19)      NOT NULL,
 CREATED              datetime         NOT NULL,
 LAST_DDL_TIME        datetime         NOT NULL,
 TIMESTAMP            varchar(19)      NOT NULL,
 STATUS               varchar(7)       NOT NULL,
 TEMPORARY2           varchar(1)       NOT NULL,
 GENERATED            varchar(1)       NOT NULL,
 SECONDARY            varchar(1)       NOT NULL,
 NAMESPACE            bigint           NOT NULL,
 EDITION_NAME         varchar(30)      NULL,
 PADDING1             varchar(4000)    NULL,
 PADDING2             varchar(3500)    NULL,
 ATTRIBUTE            varchar(32)      NULL
)
```

This table is populated with 1,729,204 rows. It is created on an 8K server and in order to ensure that each record in the table is approximately a page, two columns *PADDING1 VARCHAR(4000)* and *PADDING2 VARCHAR(3500)* were added and populated. The last column of this table is called *ATTRIBUTE VARCHAR(32)* and is populated with *NEWID()* in order to create a *UNIQUE identifier*. The *NEWID()* function generates human-readable, globally unique IDs (GUIDs). Note that there is a unique nonclustered index on this column called t_*ATTRIBUTE_UNIQUE_INDEX*:

```
CREATE UNIQUE INDEX t_ATTRIBUTE_UNIQUE_
INDEX on t (ATTRIBUTE)
```

To progress this further, we will need a mechanism to read the row (all the columns) from table t into the procedure cache. It is also important that the way we do it is best to be

done via a construct in memory. In other words, we need to loop over the index key values of table t, match them and read them. Ideally a one dimensional array of some sort could help resolve this issue. So the approach would be to load ATTRIBUTE column of table t into memory, scramble it such that ATTRIBUTE values are randomised.

ASE as of today does not offer any array handling mechanism that could be useful for this case. So in order to simulate an array, I created an in-memory one dimensional table consisting of ATTRIBUTE column in an ASE in-memory database (IMDB). We call this table *array* with the following structure:

```
SELECT ATTRIBUTE INTO array FROM t ORDER BY
NEWID()
```

Note that we do not need to create an index on this table. The table is ordered by NEWID() in order to randomise/scramble the column. We will be creating a cursor based on all ATTRIBUTE values in this table as shown in the following code:

```
CREATE PROCEDURE test_scans_with_index_sp
(
    @num_iter int=NULL
)
AS
BEGIN
 IF @num_iter IS NULL
  SET @num_iter=1
 WHILE @num_iter > 0
 BEGIN
  --
  -- Load cursor from in-memory database
  --
  DECLARE t_data_cursor CURSOR
  FOR
    SELECT
          ATTRIBUTE
    FROM ASEIMDB..array
    ORDER BY NEWID()
  DECLARE
            @OWNER                varchar(30)
          , @OBJECT_NAME          varchar(30)
          , @SUBOBJECT_NAME       varchar(30)
          , @OBJECT_ID            bigint
          , @DATA_OBJECT_ID       bigint
          , @OBJECT_TYPE          varchar(19)
          , @CREATED              datetime
          , @LAST_DDL_TIME        datetime
          , @TIMESTAMP            varchar(19)
          , @STATUS               varchar(7)
          , @TEMPORARY2           varchar(1)
          , @GENERATED            varchar(1)
          , @SECONDARY            varchar(1)
          , @NAMESPACE            bigint
          , @EDITION_NAME         varchar(30)
          , @PADDING1             varchar(4000)
          , @PADDING2             varchar(3500)
          , @ATTRIBUTE            varchar(32)

  declare @records int, @norecords int
  set @records = 0
  set @norecords = 0
  OPEN t_data_cursor
  FETCH t_data_cursor INTO
            @ATTRIBUTE

WHILE (@@SQLSTATUS=0)
BEGIN
 --
 -- Check if ATTRIBUTE exists for a given value
 --
   BEGIN
     SELECT
        @OWNER = OWNER
      , @OBJECT_NAME = OBJECT_NAME
      , @SUBOBJECT_NAME = SUBOBJECT_NAME
      , @OBJECT_ID = OBJECT_ID
      , @DATA_OBJECT_ID = DATA_OBJECT_ID
      , @OBJECT_TYPE = OBJECT_TYPE
      , @CREATED = CREATED
      , @LAST_DDL_TIME = LAST_DDL_TIME
      , @TIMESTAMP = TIMESTAMP
      , @STATUS = STATUS
      , @TEMPORARY2 = TEMPORARY2
      , @GENERATED = GENERATED
      , @SECONDARY = SECONDARY
      , @NAMESPACE = NAMESPACE
      , @EDITION_NAME = EDITION_NAME
      , @PADDING1 = PADDING1
      , @PADDING2 = PADDING2
      , @ATTRIBUTE = ATTRIBUTE
     FROM
        t
```

```
      WHERE
          t.ATTRIBUTE = @ATTRIBUTE
      END
      FETCH t_data_cursor INTO
            @ATTRIBUTE
   END
   CLOSE t_data_cursor
   DEALLOCATE CURSOR t_data_cursor

   SELECT @num_iter = @num_iter -1
 END
END
go
```

To execute this stored procedure, we will use a piece of SQL shown below with the necessary switches. These switches are explained in ASE manuals.

```
set switch on 3604  -- or use the old dbcc traceon(3604)
go
dbcc proc_cache(free_unused) -- free unused proce-
dures from procedure cache
go
set statement_cache off
go
set parallel_query off
go
print ""
declare @d1 datetime, @d2 datetime
select @d1=getdate()
SELECT * FROM master..monProcessWaits WHERE
SPID=@@SPID
EXEC DBSSD..test_SSD_reads_sp 1
SELECT * FROM master..monProcessWaits WHERE
SPID=@@SPID
select @d2=getdate()
select "Time taken in milliseconds = ",datediff(ms,@
d1,@d2)
go
exit
```

We ran the above code for databases created on *SSD (DBSSD)* and on *HDD (DBHDD)*. Note that in both cases the plan is the same (see adjacent output) and Logical IO is very similar. ASE was rebooted before each test run.

```
QUERY PLAN FOR STATEMENT 11 (at line 53).
 Optimized using Deferred Compilation.
 Optimized using Serial Mode

  STEP 1
    The type of query is SELECT.

    1 operator(s) under root

    |ROOT:EMIT Operator (VA = 1)
    |
    |  |SCAN Operator (VA = 0)
    |  | FROM TABLE
    |  | t
    |  | Index : t_ATTRIBUTE_UNIQUE_INDEX
    |  | Forward Scan.
    |  | Positioning by key.
    |  | Keys are:
    |  | ATTRIBUTE ASC
    |  | Using I/O Size 8 Kbytes for index leaf pages.

    |  | With LRU Buffer Replacement Strategy for index leaf pages.
    |  | Using I/O Size 8 Kbytes for data pages.
    |  | With LRU Buffer Replacement Strategy for data pages.

The SSD results shows as follows:
-----------------------------------     ----------
Time taken in milliseconds =     228233
```

So this job finishes in 228,233 ms or around 4 minutes running on SSD. To measure device hit ratios, I run the UNIX utility IOSTAT, because I felt that it is more accurate for short sessions than using MDA readings from *monDeviceIO* table.

```
iostat -x -d <DEVICE_NAME> 300
```

This will provide extended statistics for a given device for 5 minutes. For sake of brevity, I only display the wait time, the service time and % utilization.

- ***await***, average response time (ms) of IO requests to a device. The name is a bit confusing as this is the total response time including wait time in the requests queue (let call it qutim), and service time that device was working servicing the requests (see next column svctm). So the formula is await = qutim + svctm.
- ***svctm***, average time (ms) a device was servicing requests. This is a component of total response time of IO requests.
- ***qutim*** = await – svctm.

```
Device:  await  svctm  %util
 sdc      0.22   0.22  66.96
```

So in this instance, the service time is 0.22 ms and the queue time is 0.22 - 0.22 = 0 , i.e. no qutim for SSD. In contrast, when I run the same code on Conventional Hard drives (HDD) it finishes in 4,800,326 ms or 80 minutes! The IOSTAT run over shows:

```
Device:  await  svctm  %util
sdb       8.22   8.19  99.28
```

Now the queue time is 8.22 – 8.19 = 0.03 ms, in contrast to no queue time for SSD. On the face of it SSD looks to be 20 times faster compared to HDD primarily due to much lesser service time. So we have an increase in service time plus an increase in queue time when using HDD. In other words, SSD is very impressive for fast reads.

Scenario 2: SSD Versus HDD for Full Table Scans

Up to now we were looking at the performance of ASE on SSD vs HDD for random IOs via index scans. So how does SSD fair for full table scans? Are we going to see the same performance improvements as we just completed the test for.

We will again rely on table t. However, we will drop the index on ATTRIBUTE column and indeed any other index on this table. We will then go ahead and repopulate table array in IMDB as follows:

```
INSERT INTO array SELECT ATTRIBUTE FROM
DBSSD..t
```

Note that we have created a heap table for array with rows of ATTRIBUTE physically organised like those for table t. Since this is going to be a table scan of over 1.7 million rows, we confine ourselves to a cursor of 100 rows only. In other words, the cursor in the stored procedure above is *redefined* as:

```
DECLARE t_data_cursor CURSOR
  FOR
    SELECT  TOP 100
          ATTRIBUTE
    FROM ASEIMDB..array
```

Attention should be to the top 100 rows and the absence of any *ORDER BY* clause above. The results of this query for both SSD and HDD are shown below.

```
QUERY PLAN FOR STATEMENT 11 (at line 52).
 Optimized using Deferred Compilation.
 Optimized using Serial Mode

STEP 1
  The type of query is SELECT.

  1 operator(s) under root

  |ROOT:EMIT Operator (VA = 1)
  |
  |  |SCAN Operator (VA = 0)
  |  |  FROM TABLE
  |  |  t
  |  |  Table Scan.
  |  |  Forward Scan.
  |  |  Positioning at start of table.
  |  |  Using I/O Size 64 Kbytes for data pages.
  |  |  With MRU Buffer Replacement Strategy for data pages.
```

Not surprisingly as expected the plan is doing a table scan.

Just to clarify and diverge a bit, any table scan involves Asynchronous Prefetch Reads (APF) reads. These are displayed in the MDA table *monDeviceIO*. That means that ASE dispatches the IO request *ahead* of when it is actually needed. So the only *time* ASE has to wait for an APF is when it gets to a page that should have been there for an APF and it had not been prefetched yet. However, in almost all cases it gets the page read from the APF so fast there is no measurable time lag. The point is that the WaitTime for the process will *not* be the same as the device IO time as the process should (ideally) not be waiting on an APF, which is the whole idea of an APF to asynchronously prefetch pages so that when the SPID Qexec gets to that page, it is a logical IO as opposed to a physical IO and this has a marked impact on the overall timing of table scans. In other words, with APF the impact of timing on HDD reads is somehow mitigated. Also we should remember that IO's can be issued concurrently/overlapping – so the device IO time is an aggregate of the concurrent/overlapping times and not the elapsed time. So monDevice.IOTime is *not* WaitTime and is *not* ElapsedTime.

Table Scans and SSD

As explained before the bulk of table scan will be done through APF which will incur mostly physical reads. We will consider wait events for this case. From the MDA readings Pre and Post procedure call as follows:

- - - Pre

SPID	InstanceID	KPID	ServerUserID	WaitEventID	Waits	WaitTime
12	0	3211289	1	29	9	0
12	0	3211289	1	31	1	0
12	0	3211289	1	250	8	0

- - - Post

SPID	InstanceID	KPID	ServerUserID	WaitEventID	Waits	WaitTime
12	0	3211289	1	29	854	0
12	0	3211289	1	31	1	0
12	0	3211289	1	124	243436	1300
12	0	3211289	1	250	8	0

The WaitEvent ID 124, this is for *wait for mass read to finish* when getting the page and indicates Physical APF reads (see below). If we work out the total waits at 1 ms each (see prior discussions), we will come to a figure of 244 seconds. The device hits from the MDA table monDeviceIO shows:

Device (size)	Reads	APFReads	Writes	Total IOs	IOTime/ms	svc_t/ms
datadev01_SSD *(16000 MB)*	6884435	6883588	0	6884435	40740800	5.9

As expected, APF reads dominate the session with an average service time of 5.9 ms. The output of sp_monitor for the procedure shows:

ProcName	DBName	SPID	ElapsedTime	CpuTime	WaitTime
test_scans_with_no_index_sp	DBSSD	12	0:43:54.72	2632290	2400

PhysicalReads	LogicalReads	PacketsSent	StartTime	EndTime
6884389	57640301	1	08:38:35	09:22:30

The elapsed time of 43 minutes above is pretty consistent with my measurements of 2,573,803 ms or 42.9 minutes

Table Scans and HDD

The final step was to run the same procedure on HDD. This is where APF reads help cushion the less efficient timing from HDD. The plan as expected was the same. The readings are shown below:

- - - Pre

SPID	InstanceID	KPID	ServerUserID	WaitEventID	Waits	WaitTime
11	0	3080216	1	29	31	0
11	0	3080216	1	31	1	0
11	0	3080216	1	250	8	0

- - - Post

SPID	InstanceID	KPID	ServerUserID	WaitEventID	Waits	WaitTime
11	0	3080216	1	29	875	0
11	0	3080216	1	31	1	0
11	0	3080216	1	124	244752	1000
11	0	3080216	1	214	20	0
11	0	3080216	1	250	8	0

"Note the timings of 2,573,803 ms just under 43 minutes for SSD versus 5,572,443 ms or 92 minutes for HDD, slightly better than twice and nowhere near random reads with index scans!"

For HDD, the total WaitTime is 245,616 ms or 245 seconds. The device hits from the MDA table monDeviceIO shows:

Device (size)	Reads	APFReads	Writes	Total IOs	IOTime/ms	svc_t/ms
datadev01_HDD *(16000 MB)*	6863599	6862753	0	6863599	83557900	12.1

As expected, APF reads dominate the session with an average service time of 12.1 ms twice slower compared to the service time of 5.9 ms for SSD. Again the output of sp_monitor for the procedure shows:

ProcName	DBName	SPID	ElapsedTime	CpuTime	WaitTime
test_scans_with_no_index_sp	DBHDD	11	1:32:52.40	5570761	1600

PhysicalReads	LogicalReads	PacketsSent	StartTime	EndTime
6863709	57640301	1	01:10:28	02:43:20

Note the timings of 2,573,803 ms just under 43 minutes for SSD versus 5,572,443 ms or 92 minutes for HDD, slightly better than twice and *nowhere near random reads with index scans!*

Conclusion

In the first part of this article, we showed the use case for Solid State Disks for read activities in ASE 15. We also concluded that SSD are best suited for random access via index scans as seek time disappears and accessing any part of the disks is just as fast. When the query results in table scans, the gain is not that much. The reason being that HDD is reading sequentially from pages and the seek time is considerably less compared to random reads. In addition, APF reads contribute to the fact that the reads from HDD have a lesser impact to the overall timing of query compared to the random reads. To confirm the points I did very similar tests on Oracle Database 11g Release 2, using the same hardware in a configuration similar to ASE test with almost identical base data. This test on Oracle also confirmed the results obtained from ASE.

In part two, we'll provide the use case for Solid State Disks for the write intensive activities so please stay tuned! ■