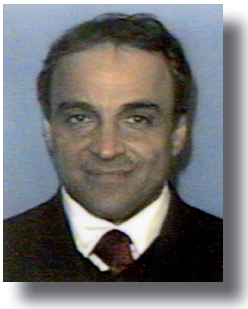
Use Case Comparison of ASE 15.7 on Solid State Disks [Part 2]

In the first part of this article we looked at the performance of Solid State Disks for read intensive activities. In this second part we will look at the performance of ASE on Solid State Disks compared to the traditional Hard Disk Drives for write intensive activities.

By Mich Talebzadeh



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.

As we learned in *"Use Case Comparison of ASE 15.7 on Solid State Disks [Part 1]"* which was published in the April 2012 edition of *ISUG Technical Journal*, database performance is ultimately constrained by the need to read and write data from a persistent storage source. For the introduction to SSDs, the rationale in using SSDs, the environment set-up and the approach we will be using, please refer to back to Part 1. The purpose of this second part is to examine the use case for Solid State Disks in terms of the performance of ASE in write intensive activities.

Scenario 1: SSD Versus HDD for Bulk Inserts

Expected Outcome

SSD will fair reasonably better compared to HDD but not that much faster. The initial seek time for SSD will be faster but the rest of the writes will be sequential extent after extent which will not be much different from what HDD does.

In Summary

- The first seek time for positioning the disk on the first available space will be much faster for SSD than HDD.
- The rest of the writes will have marginal improvements with SSD over HDD.

Test Case

We base our test case on table *t* that we introduced in Part 1. Just to recap, table *t* is populated with 1,729,204 rows of random data. 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).

Our focus is to see the time it takes to write a million rows to disk. To reduce reads from disk, we can use ASE in-memory database (IMDB) to cache this table fully in-memory and eliminate the impact of reads from disk to buffer when doing bulk inserts. Thus, for this purpose, table *t* was created and populated in IMDB. As defined in Part 1, the structure of table *t* 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
 )
```

Instrumentation

ASE MDA tables offer an excellent mechanism to measure various matrices associated with a SQL statement. In particular, the MDA table *master..monProcessActivity* will be very useful in this case. The focus of this study for both SSD and HDD would be to see:

- The volume of writes generated.
- The volume of log generated.
- The timing for bulk insert.

The general approach would be to write a script that will take the top million rows from table *t* in IMDB and put them in tables DBHDD..testwrites and DBSSD..testwrites in databases created on HDD and SSD disks respectively. The SQL will look like the following:

```
set nocount on
go
declare @dbname varchar(30)
select @dbname = db_name()
print "Database is %1!",@dbname
set switch on 3604
dbcc cachedataremove(@dbname)
go
```

```
if exists(select 1 from sysobjects where type = 'U' and
name = 'testwrites')
  drop table testwrites
go
dbcc proc_cache(free_unused) -- free unused procedures
from procedure cache
go
set statement_cache off
go
set parallel_query off
go
print ""
declare @d1 datetime, @d2 datetime
declare @ULCBytesWrittenStart int
declare @ULCBytesWrittenEnd int
declare @CPUTimeStart int
declare @CPUTimeEnd int
declare @WaitTimeStart int
declare @WaitTimeEnd int
declare @PhysicalWritesStart int
declare @PhysicalWritesEnd int
declare @PhysicalReadsStart int
declare @PhysicalReadsEnd int
declare @PagesWrittenStart int
declare @PagesWrittenEnd int
declare @PagesReadStart int
declare @PagesReadEnd int
declare @LogicalReadsStart int
declare @LogicalReadsEnd int
select @d1=getdate()
dump tran DBSSD with truncate_only
--dbcc checktable(syslogs)
SELECT
       @ULCBytesWrittenStart = ULCBytesWritten
      ,@CPUTimeStart = CPUTime
      ,@WaitTimeStart = WaitTime
      ,@PagesWrittenStart = PagesWritten
      ,@PhysicalReadsStart = PhysicalReads
      ,@PhysicalWritesStart = PhysicalWrites
      ,@PagesReadStart = PagesRead
      ,@LogicalReadsStart = LogicalReads
FROM
       master..monProcessActivity
WHERE    SPID = @@SPID
SELECT TOP 1000000 * INTO testwrites from
ASEIMDB..t
```

```
SELECT "Number of rows inserted = " ||
CONVERT(CHAR(10),@@ROWCOUNT)
SELECT
        @ULCBytesWrittenEnd = ULCBytesWritten
       ,@CPUTimeEnd = CPUTime
       ,@WaitTimeEnd = WaitTime
       ,@PagesWrittenEnd = PagesWritten
       ,@PhysicalReadsEnd = PhysicalReads
       ,@PhysicalWritesEnd = PhysicalWrites
       ,@PagesReadEnd = PagesRead
       ,@LogicalReadsEnd = LogicalReads
FROM
        master..monProcessActivity
WHERE    SPID = @@SPID
--dbcc checktable(syslogs)
SELECT "Log written/MB = "||CONVERT(CHAR(10),
CONVERT(NUMERIC(10,2),(@ULCBytesWrit-
tenEnd-@ULCBytesWrittenStart)/(1024.*1024.)))
SELECT "CPU time/ms = " || CONVERT(CHAR(10),@
CPUTimeEnd-@CPUTimeStart)
SELECT "Wait time/ms = " || CONVERT(CHAR(10),@
CPUTimeEnd-@CPUTimeStart)
SELECT "Physical reads from disk = " ||
CONVERT(CHAR(10),@PhysicalReadsEnd - @Physi-
calReadsStart)
SELECT "Physical writes = " ||
CONVERT(CHAR(10),(@PhysicalWritesEnd-@Physi-
calWritesStart))
SELECT "Pages written = " || CONVERT(CHAR(10),(@
PagesWrittenEnd-@PagesWrittenStart))
--SELECT "Physical writes/MB = " || CONVERT(C
HAR(10),CONVERT(NUMERIC(10,2),(@Physical-
WritesEnd-@PhysicalWritesStart)*@@maxpagesize/
(1024.*1024.)))
SELECT "Pages written/MB = " || CONVERT(CHA
R(10),CONVERT(NUMERIC(10,2),((@PagesWrit-
tenEnd-@PagesWrittenStart)*1.0*@@maxpagesize/
(1024.*1024.))))
SELECT "Pages Read/MB = " || CONVERT(CHAR(10)
,CONVERT(NUMERIC(10,2),((@PagesReadEnd-@Pag-
esReadStart)*1.0*@@maxpagesize/(1024.*1024.))))
SELECT "Logical Reads = " ||
CONVERT(CHAR(10),(@LogicalReadsEnd-@Logical-
ReadsStart))
exec sp_spaceused testwrites
set statistics plancost, resource, time, io off
set showplan off
```

```
select @d2=getdate()
select "Time taken in milliseconds = ",datediff(ms,@
d1,@d2)
print ""
go
```

The commented lines are there in case you want to double check the figures. A typical showplan for this type of statement is shown below:

```
STEP 1
    The type of query is CREATE TABLE.

  STEP 2
    The type of query is INSERT.

    3 operator(s) under root

    |ROOT:EMIT Operator (VA = 3)
    |
    |  |INSERT Operator (VA = 2)
    |  | The update mode is direct.
    |  |
    |  |  |TOP Operator (VA = 1)
    |  |  | Top Limit: 1000000
    |  |  |
    |  |  |  |SCAN Operator (VA = 0)
    |  |  |  | FROM TABLE
    |  |  |  | ASEIMDB..t
    |  |  |  | Table Scan.
    |  |  |  | Forward Scan.
    |  |  |  | Positioning at start of table.
    |  |  |  | Table (or index) resides in in-memory database.
    |  |
    |  | TO TABLE
    |  | testwrites
    |  | Using I/O Size 64 Kbytes for data pages.
```

Results of Bulk Inserts for HDD

The output below shows the results of the bulk inserts for the HDD table:

```
Number of rows inserted = 1000000
Log written/MB = 3.12
CPU time/ms = 41900
Wait time/ms = 41900
Physical reads from disk = 9
Physical writes = 44657
```

```
Pages written = 323316
Pages written/MB = 2525.90
Pages Read/MB = 0.07
Logical Reads = 1372446

name       rowtotal  reserved      data          index_size  unused
---------- --------- ------------- ------------- ----------- --------
testwrites 1000000   2666696 KB    2666672 KB          0 KB    24 KB

Time taken in milliseconds =     47863
```

So in summary we inserted a million rows into this table. It created 3.12 MB *log records*. You can double check this with *dbcc checktable(syslogs)* before and after the insert if you wish. As expected, there were hardly any *Physical reads* (only 9, we were reading from 8K buffer pools in IMDB).

Physical writes is the number of write operations performed. That is physical IO that can include larger pool writes. There were 44,657 of them. In this case, the insert statement was using 64K bytes for Physical writes. This will be clearer when we look at the number of *Pages written*. It shows 323,316 pages written that roughly shows 7 to 1 ratio to Physical writes indicating that (as confirmed by showplan), these were large pool 64K page writes.

We can therefore use *Pages written* and work out the volume of data. The calculated parameter *Pages Written/MB* gives 2,525.90MB as the amount of data written. The *sp_spaceused* gives the volume of data at 2,666,672KB or 2,604MB.

Although there is a slight difference between MDA readings and sp_spaceused, I would say given that there are different counters, this would be expected. The systematics are built in these matrices and as long as we compare MDA readings for HDD versus SSD, the systematics tend to cancel out. The *CPU time* was 41,900ms and the *Time taken in milliseconds* was 47,863ms.

Results of Bulk Inserts for SSD

The output below shows the results of the bulk inserts for the SSD table:

```
Number of rows inserted = 1000000
Log written/MB = 3.14
CPU time/ms = 19500
Wait time/ms = 19500
Physical reads from disk = 7
Physical writes = 44744
Pages written = 323354
Pages written/MB = 2526.20
Pages Read/MB = 0.05
Logical Reads = 1373579

name       rowtotal  reserved      data          index_size  unused
---------- --------- ------------- ------------- ----------- --------
testwrites 1000000   2666736 KB    2666672 KB          0 KB    64 KB

Time taken in milliseconds =     21303
```

The volume of Log Written is 3.14MB for million rows. Physical reads are 7 and the Pages written is 2,526.20MB. The remaining stats are almost identical when we ran the test for HDD table. CPU time in this case is 19,500ms and Time taken in milliseconds is 21,303ms.

So compared to bulk insert on HDD, we are not generating any more data or log. The only improvement is the CPU time at 19,500ms compared to HDD CPU time at 41,900ms. This shows that bulk inserts on SSD are around twice as fast compared to the same inserts on HDD. So unfortunately SSD does not really write much faster than the spinning magnetic disks, especially for the type of writes (sequential) that we do with bulk inserts.

Scenario 2: SSD Versus HDD for Updates

We now turn our attention to random updates and see how Solid State Disks fair compared to magnetic Hard Disks. Prior to doing the tests, we need to look at the way updates are done on Solid State Disks.

Solid State Disks and Updates

As mentioned in Part 1, SSD's base memory unit is a cell, which holds 1 bit in SLC and 2 bits in MLC. Cells are organized in pages (usually 4k) and pages are organized in blocks (512K). Data can be read and written in pages, but it is always deleted in blocks.

In SQL an update is a delete followed by an insert. This translates to what is known as erasing and writing at the disk level. With SSD updates, erasing is slow because even if you want to delete one page, the SSD can only delete the entire block. The controller needs to read the entire block, erase everything and write back only the bits you want to keep. To keep the re-write overhead low, manufacturers use several techniques:

- They use every cell before resorting to deleting existing cells.

- They will over-provision the SSD (i.e. build 250GB SSD but only show the OS 225GB), so writes can be completed quickly using the "spare space" and then the required deletes can be completed asynchronously in the background.

The outcome of this is that it requires the SSD controller to be pretty smart and maintain free lists of pages that can be used for writing, and pages that need cleaning. This is one of the major differences between different SSD devices – how well the controller manages the writing and erasing cycles on the device.

Note that erasing being slower in SSD does not mean that SSD will perform worse compared to HDD. When doing random updates via index scan, SSD speed, in using an index to locate the rows, is much higher than HDD. What we are alluding here is that the speed of random writes to SSD is not on par with the speed of random reads from SSD.

Performing Updates on HDD

To perform updates via an index, we will need to add an index to table testwrites. The column OBJECT_ID is monolithically increasing and thus unique. So we will just create a unique index on this column on table *testwrites*:

```
CREATE UNIQUE INDEX testwrites_ui ON testwrites
(OBJECT_ID)
go
UPDATE INDEX STATISTICS testwrites
go
```

In order to do our test for updates, we will deploy the following script:

```
set nocount on
go
declare @dbname varchar(30)
select @dbname = db_name()
print "Database is %1!",@dbname
set switch on 3604
dbcc cachedataremove(@dbname)
go
dbcc proc_cache(free_unused) -- free unused procedures
from procedure cache
go
set statement_cache off
go
set parallel_query off
go
declare @d1 datetime, @d2 datetime
declare @ULCBytesWrittenStart int
declare @ULCBytesWrittenEnd int
declare @CPUTimeStart int
declare @CPUTimeEnd int
declare @WaitTimeStart int
declare @WaitTimeEnd int
declare @PhysicalWritesStart int
declare @PhysicalWritesEnd int
declare @PhysicalReadsStart int
declare @PhysicalReadsEnd int
declare @PagesWrittenStart int
declare @PagesWrittenEnd int
declare @PagesReadStart int
declare @PagesReadEnd int
declare @LogicalReadsStart int
declare @LogicalReadsEnd int
select @d1=getdate()
dump tran DBSSD with truncate_only
--dbcc checktable(syslogs)
SELECT
      @ULCBytesWrittenStart = ULCBytesWritten
     ,@CPUTimeStart = CPUTime
     ,@WaitTimeStart = WaitTime
     ,@PagesWrittenStart = PagesWritten
     ,@PhysicalReadsStart = PhysicalReads
     ,@PhysicalWritesStart = PhysicalWrites
     ,@PagesReadStart = PagesRead
     ,@LogicalReadsStart = LogicalReads
FROM
      master..monProcessActivity
WHERE   SPID = @@SPID
UPDATE testwrites
SET PADDING1 = 'y'+space(3998)+'y'
FROM testwrites
WHERE OBJECT_ID BETWEEN 10000 AND
200000
SELECT "Number of rows updated = "
||CONVERT(CHAR(10),@@ROWCOUNT)
SELECT
      @ULCBytesWrittenEnd = ULCBytesWritten
     ,@CPUTimeEnd = CPUTime
     ,@WaitTimeEnd = WaitTime
     ,@PagesWrittenEnd = PagesWritten
     ,@PhysicalReadsEnd = PhysicalReads
     ,@PhysicalWritesEnd = PhysicalWrites
```

```
      ,@PagesReadEnd = PagesRead
      ,@LogicalReadsEnd = LogicalReads
FROM
      master..monProcessActivity
WHERE   SPID = @@SPID
--dbcc checktable(syslogs)
SELECT "Log written/MB = "||CONVERT(CHAR(10),
CONVERT(NUMERIC(10,2),(@ULCBytesWrit-
tenEnd-@ULCBytesWrittenStart)/(1024.*1024.)))
SELECT "CPU time/ms = " || CONVERT(CHAR(10),@
CPUTimeEnd-@CPUTimeStart)
SELECT "Wait time/ms = " || CONVERT(CHAR(10),@
CPUTimeEnd-@CPUTimeStart)
SELECT "Physical reads from disk = " ||
CONVERT(CHAR(10),@PhysicalReadsEnd - @Physi-
calReadsStart)
SELECT "Physical writes = " ||
CONVERT(CHAR(10),(@PhysicalWritesEnd-@Physi-
calWritesStart))
SELECT "Pages written = " || CONVERT(CHAR(10),(@
PagesWrittenEnd-@PagesWrittenStart))
--SELECT "Physical writes/MB = " || CONVERT(C
HAR(10),CONVERT(NUMERIC(10,2),(@Physical-
WritesEnd-@PhysicalWritesStart)*@@maxpagesize/
(1024.*1024.)))
SELECT "Pages written/MB = " || CONVERT(CHA
R(10),CONVERT(NUMERIC(10,2),((@PagesWrit-
tenEnd-@PagesWrittenStart)*1.0*@@maxpagesize/
(1024.*1024.))))
SELECT "Pages Read/MB = " || CONVERT(CHAR(1
0),CONVERT(NUMERIC(10,2),((@PagesReadEnd-@
PagesReadStart)*1.0*@@maxpagesize/(1024.*1024.))))
SELECT "Logical Reads = " ||
CONVERT(CHAR(10),(@LogicalReadsEnd-@Logical-
ReadsStart))
exec sp_spaceused testwrites
set statistics plancost, resource, time, io off
set showplan off
select @d2=getdate()
select "Time taken in milliseconds = ",datediff(ms,@
d1,@d2)
print ""
go
```

Note again that where the code is commented out, you can use them to double check the validity of the results. Again, we reboot ASE before the run.

What we are doing in here is updating the base table for the column PADDING1 which is 4000 bytes. The search is done via the unique index testwrites_ui on testwrites table. A typical showplan shows the following:

```
STEP 1
    The type of query is UPDATE.

    3 operator(s) under root

    |ROOT:EMIT Operator (VA = 3)
    |
    |  |UPDATE Operator (VA = 2)
    |  | The update mode is deferred_varcol.
    |  |
    |  |  |RESTRICT Operator (VA = 1)(5)(0)(0)(0)(0)
    |  |  |
    |  |  |  |SCAN Operator (VA = 0)
    |  |  |  | FROM TABLE
    |  |  |  | testwrites
    |  |  |  | Index : testwrites_ui
    |  |  |  | Forward Scan.
    |  |  |  | Positioning by key.
    |  |  |  | Keys are:
    |  |  |  |  OBJECT_ID ASC
    |  |  |  | Using I/O Size 64 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.
    |  |
    |  | TO TABLE
    |  | testwrites
    |  | Using I/O Size 8 Kbytes for data pages.
```

It scans the index using large 64KB I/O pools to find the rows and updates the rows using 8KB for data pages. Needless to say, the updates are always done in the cache and written back to disks. The results are shown below:

```
Number of rows updated = 77746
Log written/MB = 584.33
CPU time/ms = 607400
Wait time/ms = 607400
Physical reads from disk = 189643
Physical writes = 1222
```

```
Pages written = 2438
Pages written/MB = 19.04
Pages Read/MB = 2268.33
Logical Reads = 727841

name       rowtotal  reserved      data          index_size   unused
---------- --------- ------------- ------------- ------------ --------
testwrites 1000000   3258296 KB    3243360 KB    14800 KB     136 KB

Time taken in milliseconds =     966466
```

The above statement resulted in 77,746 rows being updated. The update created around *Log written/MB* = 584.33MB of log. Note that this time we have *Physical reads from disk* = 189,643 and *Logical Reads* = 727,841. This is not surprising as the selected pages have to be read from the disk to the buffer, updated and written back to disk. It resulted in *Physical writes* = 1,222 and that corresponded to *Pages written* = 2,438. Using *Pages written*, the volume of update works out to be *Pages written/MB* = 19.04MB. The update took *CPU time* = 607,400ms. The whole process took *Time taken in milliseconds* = 966,466ms to complete. The important parameters are the completion time, the number of Physical Writes and the volume of Pages written.

Performing Updates on SSD

The next task is to run the same query against Solid State Disks. The query plan shows the same but the figures for various parameters are different as shown below:

```
Number of rows updated = 77746
Log written/MB = 584.33
CPU time/ms = 34900
Wait time/ms = 34900
Physical reads from disk = 189638
Physical writes = 21691
Pages written = 43368
Pages written/MB = 338.81
Pages Read/MB = 2267.06
Logical Reads = 727841

name       rowtotal  reserved      data          index_size   unused
---------- --------- ------------- ------------- ------------ --------
testwrites 1000000   3258328 KB    3243360 KB    14800 KB     168 KB

Time taken in milliseconds =     47920
```

With SSD we have the same number of rows updated. The value of Log written is 484.33MB, the same for HDD.

"Despite this larger volume of updates, it took only 47,920ms to update the rows with SSD, 20 times faster compared to the 966,466ms that it took to update the same rows with HDD."

The number of Physical reads from disk is 189,638 almost identical to the one for HDD. Our focus is now on Physical writes and Pages written. With SSD we have Physical writes = 21,691 I/Os and Pages written = 43,368 that translates to Pages written/MB = 338.81MB of data written to disk for updates. Contrast this to the previous figures for HDD. The figure for Physical writes was 1,222 and Pages written was 2,438.

In a nutshell with SSD we are writing almost 18 times more pages compared to HDD. That may not come as a surprise as I mentioned that for updates on SSD, the erase is done in blocks. Each block of 512K consists of 128 * 4K pages. Our server is built on 8K page. So each OS block contains 64 ASE pages. Taking ratios of 43,368 pages and 64 pages in a block, the update operation on SSD resulted in 43368/64 ~ 678 SSD blocks each at 512K to be erased and re-written. In contrast with HDD, there were only 2,438 pages were updated. With conventional magnetic disk, a block size in our case is 4K. So in total we updated 4,876 conventional 4K OS blocks.

Despite this larger volume of updates, it took only 47,920ms to update the rows with SSD, 20 times faster compared to the 966,466ms that it took to update the same rows with HDD.

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.

In this second part of this article we looked at write intensive activities. We proved that for pure inserts into a table the gain in performance with SSD is around twice compared to HDD. With random updates via index scan, we saw higher write volumes for SSD compared to HDD due to block erase by SSD. Although this made an impact in increasing the volume of writes, the gain in performance with SSD is much higher compared to that of HDD. ■