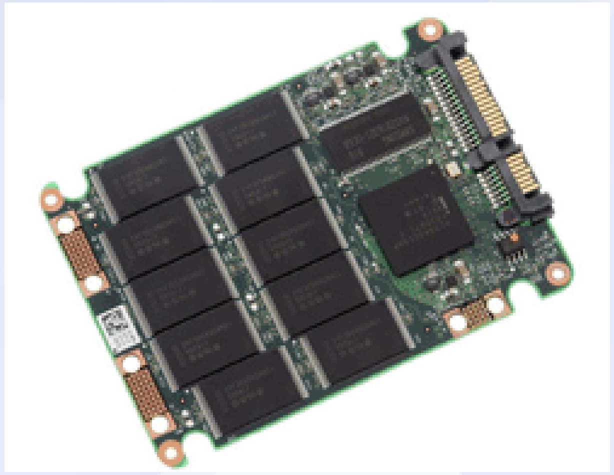
SSDs vs HDDs for DBMS

by Glen Berseth
York University, Toronto



- So slow
- So cheap
- So heavy

- So fast
- So expensive
- So efficient

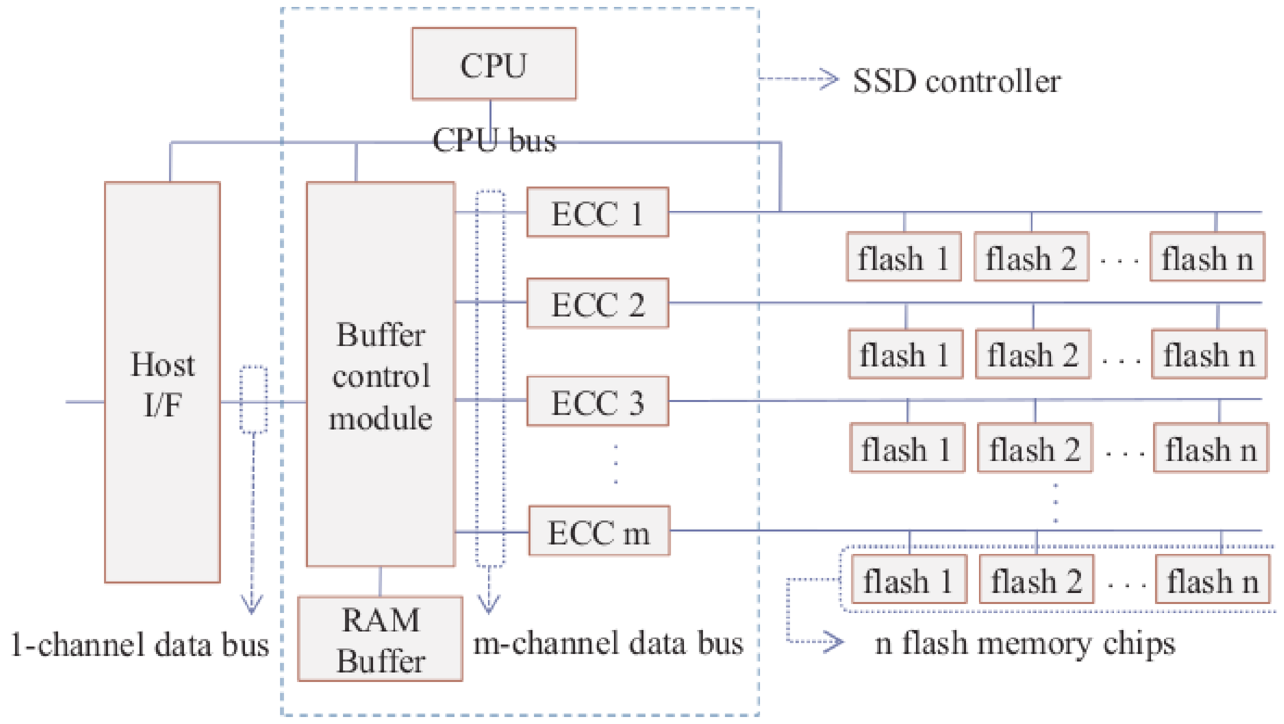
NAND based flash memory

- Retains memory without power
- It works by trapping a small amount of energy that would signify a 1 at that memory address
- NAND flash is very difficult to produce
  - NAND is generally rather large. This is to make it very reliable and durable so that the electronic charge can be held with little leakage
  - The smaller the NAND transistors become the more prone they are to wear
- Smaller power requirement, quiet and resistant to physical shock
- DRAM SSDs??

SSD Structure

- A small amount of SRAM is used to hold code and used for buffering
- SSDs have *super-blocks* that are physically seperate blocks on mapping scheme is based
- SSDs have interleaved access to super-blocks. This is similar to how RAID arrays work
- The address mapping scheme is based on super-blocks in order to limit the amount of information required for logical-to-physical addressing
- A super-block can consists of eight erase units (or large blocks) of 128 KBytes or more each
- Latency 0.2msec for read and 0.4 msec for write

Solid State Drive Internals



Internal architecture of flashSSDs

Characteristics of SSDs

- Uniform random access speed
- Accessing data is almost proportional to the amount of data accessed
- Writing is expensive. Writing can only be done to an empty space, it is costly to erase data
  - Erasing data can only be done in *erase units*, typically > = 128 kbytes
- Assymetric speed (reading is ~ twice as fast as writing)
- NAND is truly slow. Speed scales with the number of parrallel NAND flash chips used on the device

Caviets of SSDs

Storage	Hard Disk	Flash SSD
Average latency	8.33 msec	0.2 ms(read) 0.4ms(write)
Sustained Transfer rate	110 MB/sec	56 MB/sec (read) 32 MB/sec (write)

- SSDs have algorithms in the background that are garbage collecting and performing wear-leveling

- Can be a problem if accessing the same super-block or flash block repetidly
- Does not have true random write
- More susceptible to firmware bugs

Writing to SSDs

- Writing is very expensive on an SSD
- It is not possible to overwrite data in an SSD (no in-place update)
    - Information can only be written to a empty spot on the drive (1s can only be changed to 0s)
- To create areas that can be written, the drive will store the data else wear and then garbage collection with go back and remove the dirty data to create free space
- As mentioned before, data can only be erased in super blocks which are 128kbytes or greater on the drive. This becomes an issue if there is data on that superblock that is still clean
- Because of this SSDs do not generally store data sequentially, it gives the device no advantage
    - Remember wear leveling

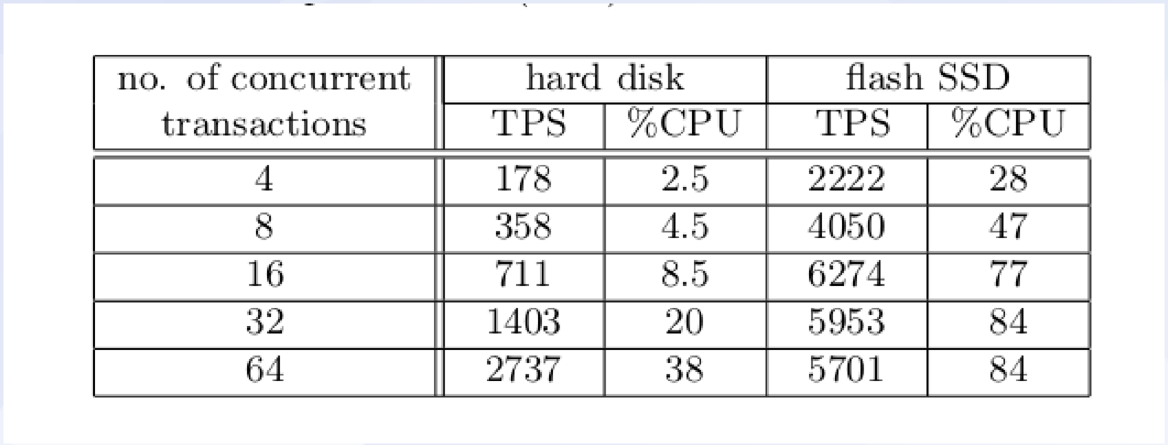
Now onto comparisons

Transaction log

- The transaction log is an audit of the updates that have been done to the database so the database can recover from unexpected faults
- Commits to the transaction log are a bottleneck of a commercial database system
- Although many transactions in a database are asynchronous, the transaction log must force-write commit log entries to the tail of the log for durability of the database
- Often, the transaction log is stored on a separate disk to further avoid this bottleneck and for reliability

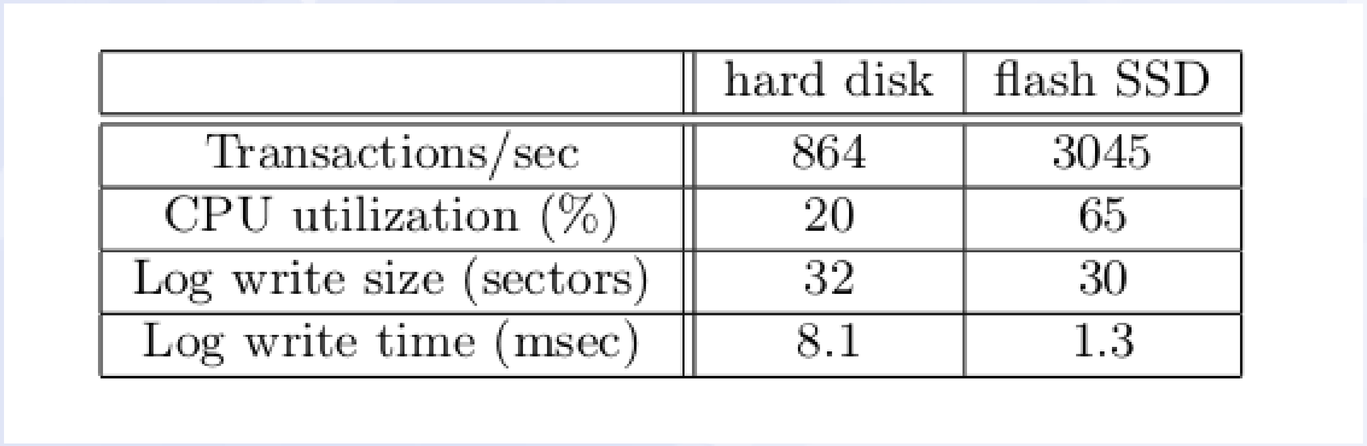
SSD and HDD comparisons (TPS)

- Embedded SQL program
  - Multi-threaded to simulate concurrent transactions
  - Each thread updates a single record and commits, and repeats this cycle of update and commit continuously
  - The entire table data were cached in memory so that the focus of the experiment was to test the transaction log forced write efficiency
  - Units are messured in Transactions Per Second (TPS)



no. of concurrent transactions	hard disk		flash SSD	
	TPS	%CPU	TPS	%CPU
4	178	2.5	2222	28
8	358	4.5	4050	47
16	711	8.5	6274	77
32	1403	20	5953	84
64	2737	38	5701	84

TCP-B



	hard disk	flash SSD
Transactions/sec	864	3045
CPU utilization (%)	20	65
Log write size (sectors)	32	30
Log write time (msec)	8.1	1.3

- Again, more than enough memory was allocated to allow the entire table to be stored in memory
- Difference in log write size is explained by group commit
- SSD throughput is 3.5 times that of HDD which is a result of decreased log write time
- Also seen is the increased utilization of CPU. This is because the SSD transactions are shorter, in turn locks on resources are released quickly

Transactions HDD

- Transaction can be in one of the three distinct states. Namely, a transaction:

(1) is still active and has not requested to commit

(2) has already requested to commit but is waiting for other transactions to complete forced-writes of their log records, or

(3) has requested to commit and is currently force-writing its own log records to stable storage

- When a HDD is used to store the data, there is inceased latency of disk writes. This leads to many transactions kept in the second and third state. This is the reason transaction throughput is low for HDD even for concurent transactions

Transactions SSD

- Notice the CPU usage of the computer when using a SSD
- Because of the much smaller latency of SSDs, transactions were not kept in the second and third state
- The focus of concurrent transactions benefits the architecture of the SDD
- The efficiency of the transactions peeked because of the much higher utilization of the SDD, showing that the CPU became a limiting factor

CPU factor

- The curve for the HDD rises steadily with the number of concurrent transactions
- The curve for the SSD rises quickly and when the saturation point of the CPU is reached the number of transactions levels off
- This shows that the CPU is not a limiting factor until the saturation point is reached in the evaluation of the different storage mediums

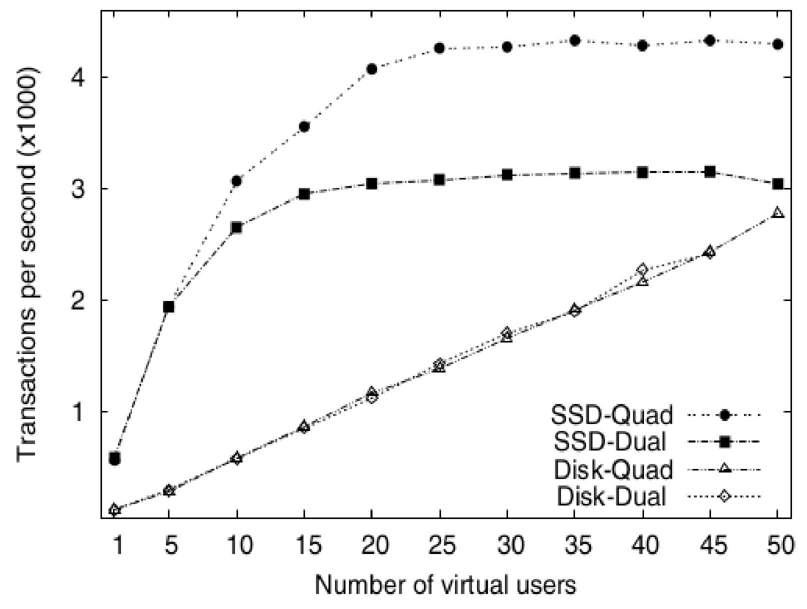


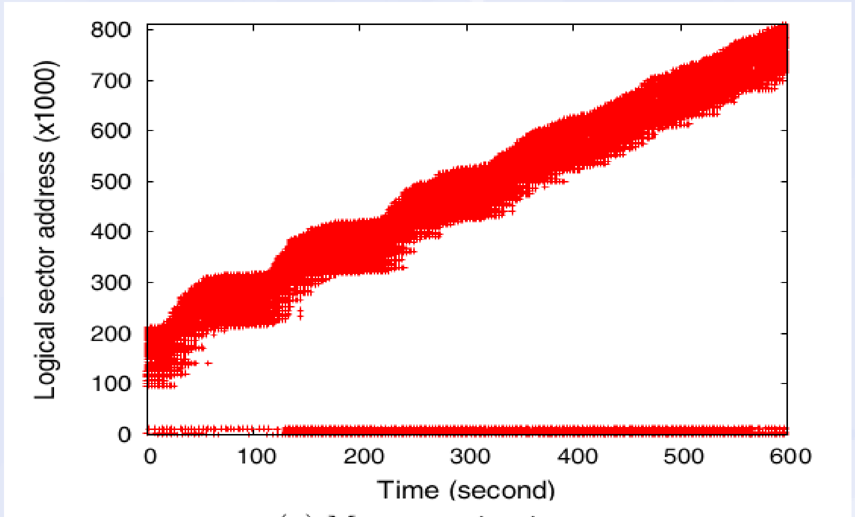
Figure 3: Commit-time performance of TPC-B benchmark : I/O-bound vs. CPU-bound

- "-Quad" is the same experiment but with using a quad core CPU instead of a dual core

Multi-Version Concurrancy Control (MVCC)

- Traditional concurrent systems are based on locks
- Read consistency is supported by providing multiple versions of a data object minimizing the number of locks needed
- However to support updates, versions of objects are stored in *rollback segments*
- Rollback segments are usually on stable storage to support the different images of the data
- Writing an object involves writing its before image to a rollback segment in addition to writing undo and redo log records
- Reading an object when using MVCC can be more costly
- When reading an object, the system needs to check if the object has been updated by other transactions, and needs to fetch an old version from a rollback segment

MVCC Write Performance

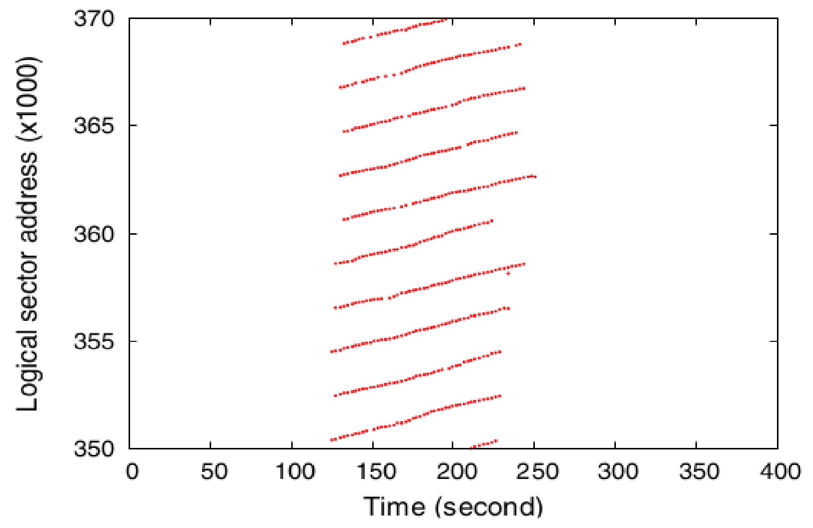


(a) Macroscopic view

- NOTE: The bottom of this figure shows the inplace updating of meta data related to the rollback segments

- The rollback segments were created in a seperate disk drive which stored nothing but the rollback segments
- This figure shows the pattern of writes we observed in the rollback segments of a commercial database server processing a TPC-C workload
- Average time for writing a block to a rollback segment was 7.1 msec for disk and 6.8 msec for flash memory SSD

MVCC Write Performance Cont.



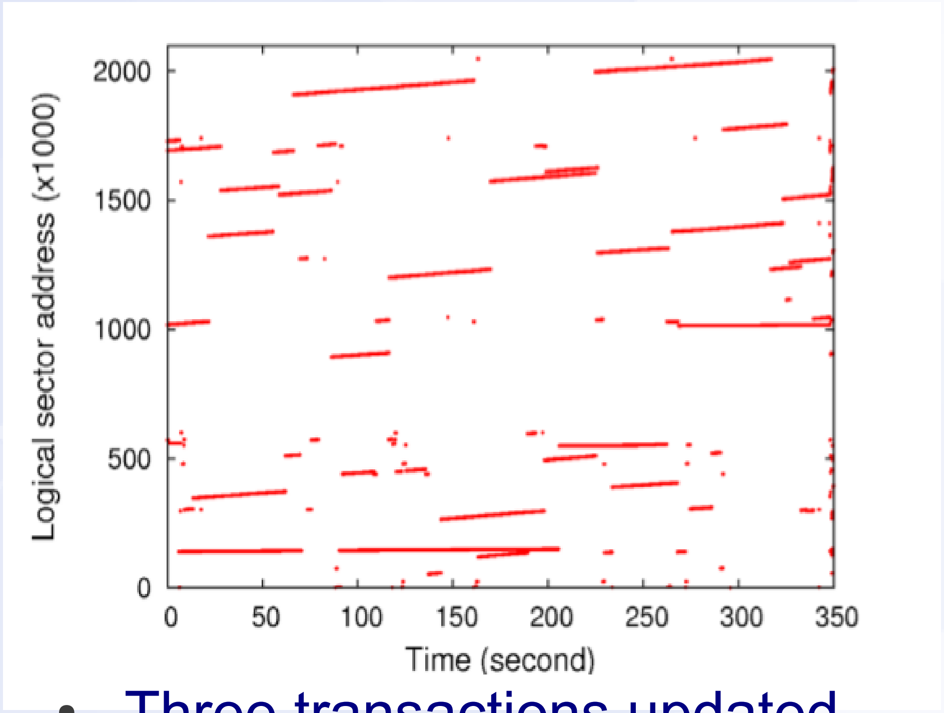
(b) Microscopic view

- If a hard disk drive were used as storage for rollback segments, each write request to a rollback segment would likely have to move the disk arm to a different track. Therefore, the cost of recording rollback data for MVCC would be significant due to excessive seek delay of disk

- Each line segment spanned a seperate logical address space equivalent to ~ one Mbyte
- This is because one Mbyte was allocated every time a rollback segment ran out of space on the current extent
- Becuase SSDs do not suffer from mechanical latency, MVCC would benifit from them greatly. Also, because the rollback segments are written in append-only-fashon, the *no-inplace-update* limitation of SSDs has no negative effect
- NOTE: Can be an issue for SSDs if no free blocks are available

MVCC Read Performance

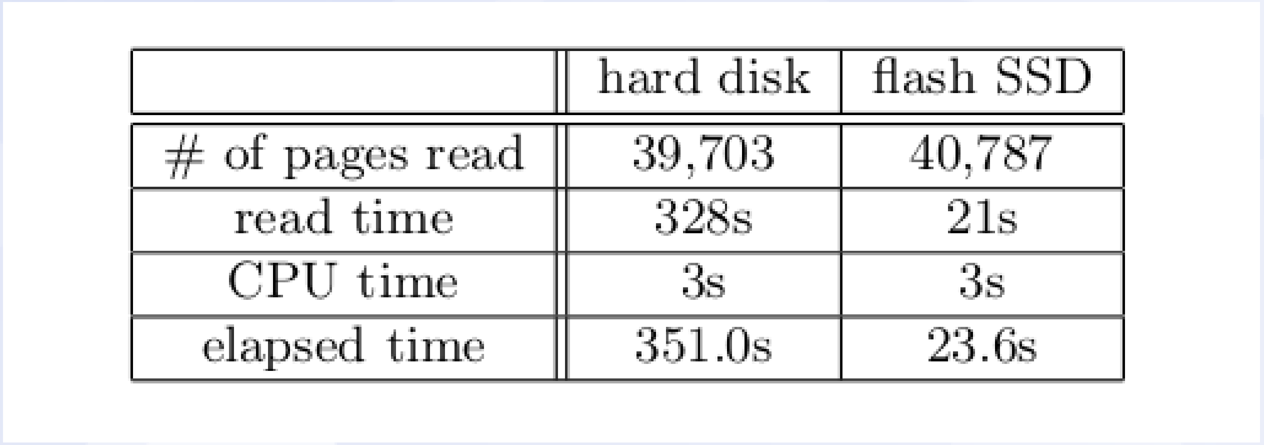
- MVCC causes an increased amount of I/O for read consistency
- Read pattern is random
- The correct version must be fetched from one of the rollback segments belonging to the transactions that updated the data object
- To understand the performace issues, an experiment that focused on the MVCC reads was done



- Three transactions updated every tuple in the table and then one transaction performed a full table scan
- Clustered but randomly scattered across one Gbyte

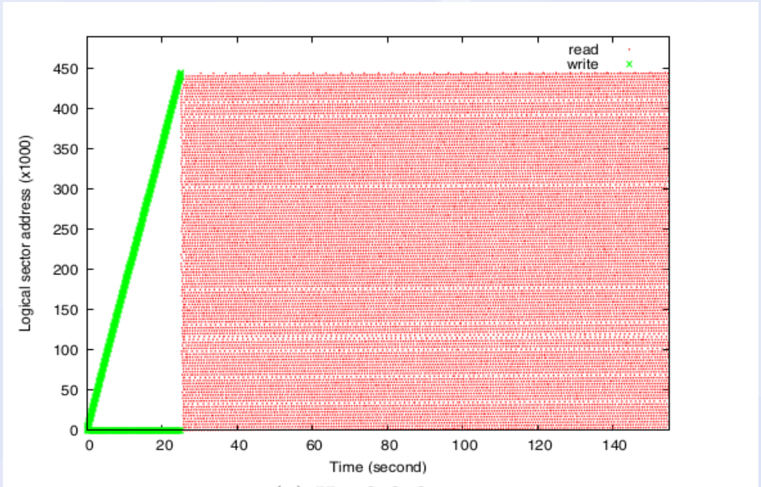
MVCC read performace continued

- Because of the random read accesses, the SSD performs far better
- The table was scanned 3 full times in the experiment. This is because the data that the full scan sought was the original data from the table
- NOTE: CPU time is the same

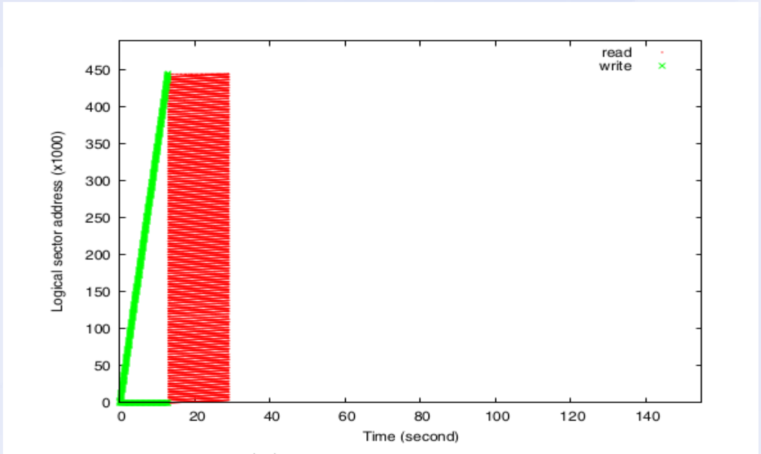


	hard disk	flash SSD
# of pages read	39,703	40,787
read time	328s	21s
CPU time	3s	3s
elapsed time	351.0s	23.6s

External sorting



(a) Hard disk

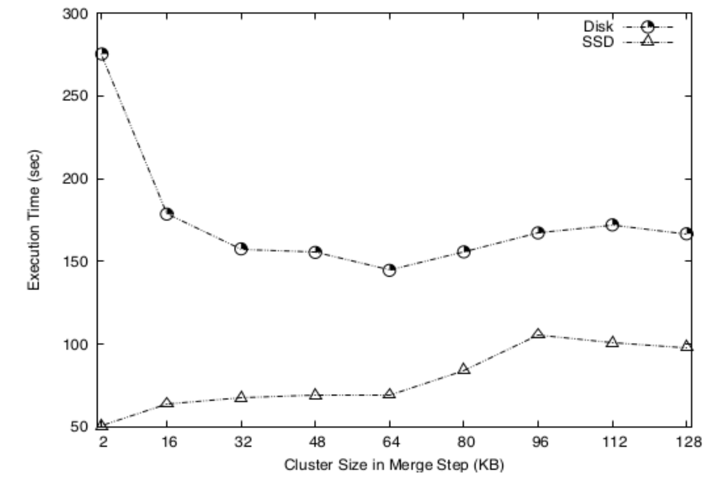


(b) Flash memory SSD

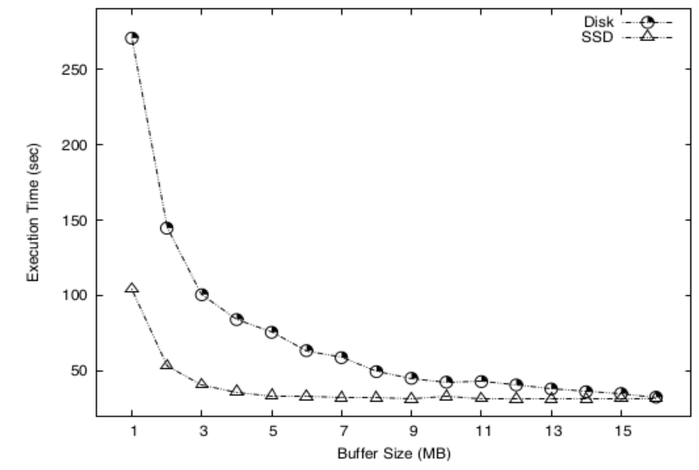
- Sequential write (for writing sorted runs) followed by random read (for merging runs)
- Sorting approximately 2,000,000 tuples
- Flash memory is faster at random reads
- SSD and HDD are comparable on sequential writes
- A trace of I/O requests

External Sorting

- Does I/O unit size have an effect?
- Sequental read because of clustered B+ tree
- First table is a comparision on varying block size. **SSDs have a limited *super-block* size**
- Increasing the total buffer size reduces the number of I/O operations and increases the number of sequential reads
- Notice that the optimal block size for SSDs appears to be 2-4kbytes



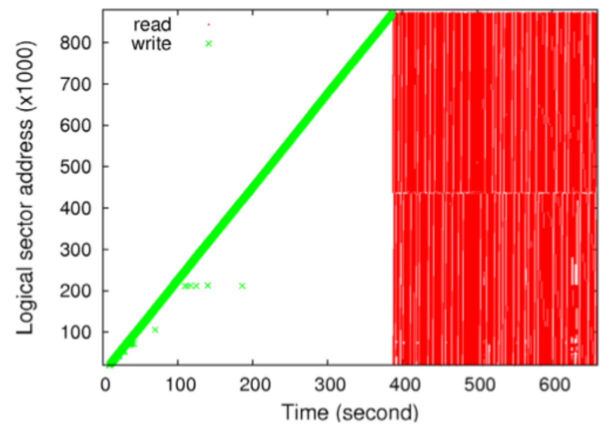
(a) Varying cluster size
(buffer cache size fixed at 2 MB)



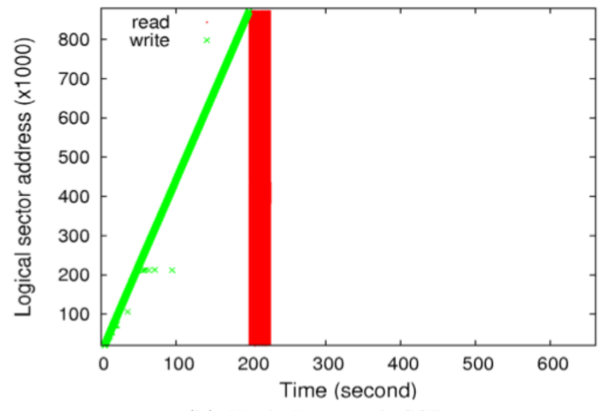
(b) Varying buffer size
(cluster size fixed at 64 KB for disk and at 2 KB for SSD)

Hash Join

- While the dominant I/O pattern of sort is sequential write (for writing sorted runs) followed by random read (for merging runs), the dominant I/O pattern of hash is said to be random write (for writing hash buckets) followed by sequential read (for probing hash buckets)
- However, this looks like sequential write and random read
- This become favorable for SSDs



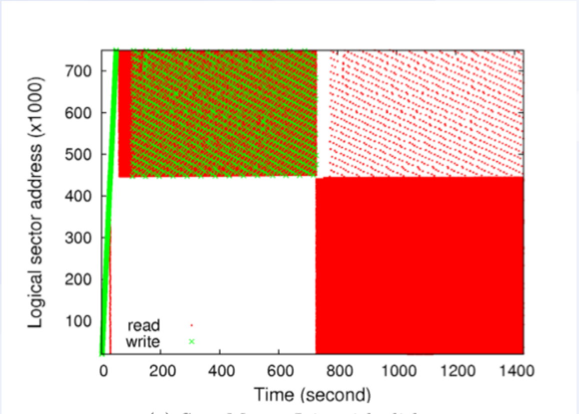
(a) Hash Join with disk



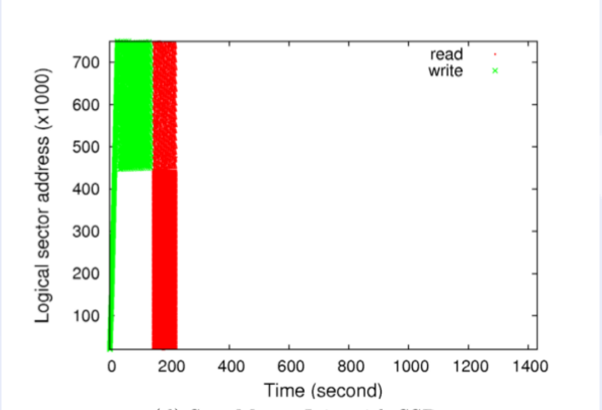
(b) Hash Join with SSD

Sort-Merge Join

- Sort-Merge Join is thought to maybe be obsolete but here for SSDs it performs similarly to Hash Join
- Points toward revisiting the sort-merge join algorithm



(c) Sort-Merge Join with disk



(d) Sort-Merge Join with SSD

Evaluating example

- SSD
  - Read Delay 1msec, Write delay 2.3 msec
  - Read time 4kb in 2 sec
  - Write time 4kb in 2.7 sec
- Example 2 (more buffer pages)
  - I/O cost per run
  - 1 + (2 x 320) + 2 + (2.7 x 320)
  - = 641 + 866 = 1507
  - I/O cost
  - Reads 2 x (2 + 1) 10,000,000
  - = 60,000,000
  - Writes 2 x (2.7 + 2.5) 10,000,000
  - = 104,000,000
  - Total = 162,000,000

Evaluating example

Example two (larger buffers)

- Read cost 1 + (2 X 16 pages) = 33
- Each pass 10,000,000 / 16 = 625,000 passes
- Cost of reading is 4 x 625,000 X 33 = 82,500,000
- Write cost 10,000,000/64 x (2 + (2.7 x 64)) x 4 = 109,250,000
- Total = 191,750,000

B+ trees

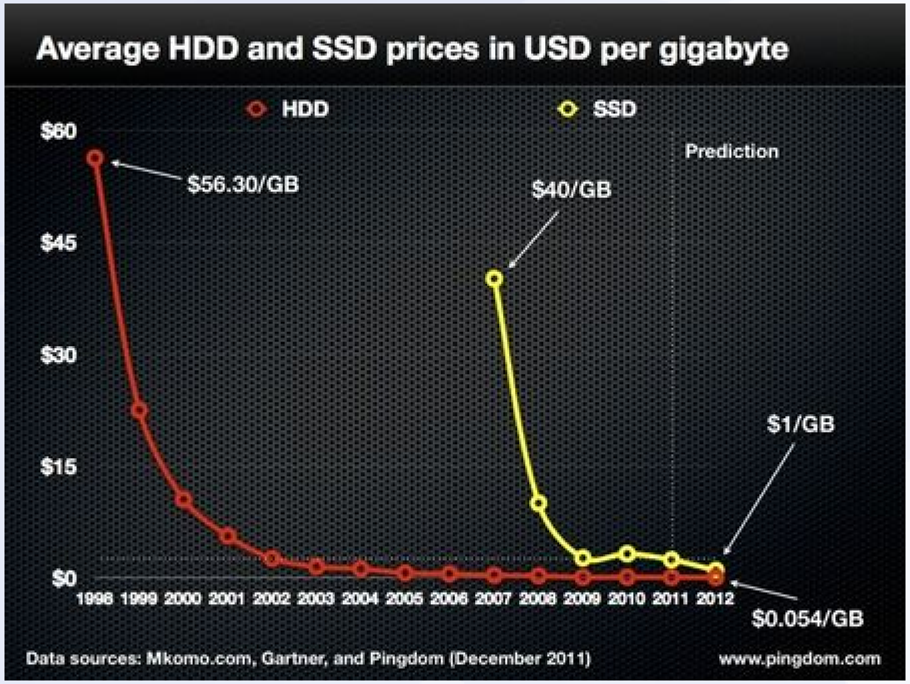
- Because NAND flash based SSDs do not support completely random writes, how do B+ trees work?
- This would make maintaining a clustered B+ tree a true headache. Then again, clustering data is obsolete when using an SSD

A few thoughts

- What hapens when your records are 1/2 the SSD super-block size + 1?
- NAND flash does have a limited number of writes it can handle
  - The more space for data on the drive, the smaller the NAND cells and the more volitile they can be

Price points

- Current
  - HDD
  - $100 CAD / 2TB
  - $0.05/ gb
  - SSD
  - $100 CAD / 120gb
  - 0.83/ gb
  - > 16 x more expensive
- SSD price does not scale



References

- C.-H. Wu and T.-W. Kuo. An adaptive two-level management for the flash translation layer in embedded systems. In Proceedings of ICCAD ’06, 2006.
- S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In SIGMOD, 2008.
- Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. Proc. VLDB Endow. 5, 4 (December 2011), 286-297.