BangDB – *Benchmark*

www.iqlect.com

White Paper - Benchmark

Overview

The purpose of the performance analysis of BangDB under few scenarios is to present a high level benchmark figure which may help users to easily map their use cases and understand what to expect from BangDB. The performance measurement is done on commodity hardware without doing any customization

These performance numbers will vary depending upon the configuration of machine, OS, size of key and value and other parameters, hence users may see different metrics when they run on their own machines in different settings. However, the benchmark shown here and the comparison of numbers with few other dbs in next section would help user to take decision in some fashion

Following machine (commodity hardware) used for the test;

• Model	: 4 CPU cores, Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, 64bit
• CPU cache	: 6MB
• OS	: Linux, 3.0.0-17-generic, Ubuntu, x86_64
• RAM	: 8GB
• Disk	: 500GB, 7200 RPM, 16MB cache
• File System	: ext4

The BangDB configuration;

• Key size	: 24 bytes - Random
• Val size	: 100 - 400 bytes – Randomly Picked
• Page size	: 8KB
• Write ahead log	: ON
• Log split check	: ON, check every 30 ms
• Buffer pool size	: 1 GB
• Background log flush	: every 50 ms
• Checkpointing	: every ~4 sec
• Buffer pool workers	: ON, every 60 ms
• Number of threads	: 4
• Volume of data written	: 32 MB – 320 MB (without compression)
• Volume of log written	: 56 MB – 520 MB (if Log = ON) (without compression)

Other parameters;

- Write and Read : Random, continuous
- Num of ops : 100K to 10M

This configuration ensures that the db runs in conservative mode where if process or db crash happens, at restart the db will recover to the point where the db crashed. There are many workers who are ensuring that the mechanisms for write ahead logging and buffer pool health. Note that in the table the data with the Log = ON depict the numbers for above configuration. However if we

switch off the log and just work with everything else as it is(apart from log and related stuff) then the numbers would look like as given in the column for Log = OFF
The tests are done in various conditions;

1. Normal: No Disk Ops

   The buffer is allocated sufficiently and all the reads and writes will happen from the pool itself. The db will not go to the disk for any reason except the continuous log flush. This ensures that the performance data reflects the true performance for the db. In the real world this condition is realistic when we use db as in memory cache only for example session data. Otherwise for persistent database, it is not possible to guarantee this

2. Stressed: Overflow to disk

   The buffer allocated is less than the amount of operations that would take place. However, the buffer allocated is around half the amount of data to be written or read. Since all operations are random and continuous, hence it will be very difficult to flush out right set of pages and bringing other right set of pages in the buffer pool. In the real world operations are less random and also it's not continuous 100 percent read and write. But this models the real world in best possible manner slightly on the conservative side. It's important to note and we will see in the performance report that BangDB performs well in this condition too

3. Totally Stressed

   The buffer allocated is much less than (around 5-10 times) the data that will be read and written into the database continuously and in random fashion. The flushing of dirty pages to reclaim some free ones makes the db to almost continuously write pages to the hard disk. The amount of read activities from disk are also high for both read and write ops. It becomes very very difficult to anticipate which pages to flush and read. The BangDB takes extra efforts to ensure that performance doesn't degrade to the level where writing to disk halts the whole system. The results show that performance degrades gracefully with the rise of amount of data vs buffer pool capacity ratio

Various sub sections of the benchmark will dig deeper into the performance analysis, please have a look at them to check out how BangDB performs in different conditions. There is a competition analysis also done to just compare BangDB with Oracle's Berkley DB and Google's Level DB. The comparison is to learn the conditions and scenarios where individual dbs performs in some manner

The average values for the throughput(ops/sec) over the 100K – 1M operations are;

Index (Access Methods)	LOG – ON		LOG - OFF	
	Write (ops/sec)	Read (ops/sec)	Write (ops/sec)	Read (ops/sec)
Btree	475,000	1,025,000	685,000	1,045,000
Ehash	500,000	1,690,000	790,000	1,675,000

Concurrency

BangDB is highly concurrent database. This means that with more number of CPUs, BangDB will tend to perform better. The most suitable spot for BangDB is to run around as many threads as number of CPU on the machine. Since performance was one of main design items for the BangDB, hence it was realized that the db has to be concurrent and should take advantage of number of CPUs in the machine. The Concurrency definitely adds the complexity and overhead but to settle with low performance even on higher capacity machine was not the intention.

BangDB achieves high concurrency mainly for the following reasons;

- The access methods (index) are highly concurrent

BangDB implements variant of Btree which is B+ link tree. The structure maintains neighboring links to the pages at the same level. The locking granularity is at the page level. During a write operation, in the worst case scenario, db locks two pages at a time. However, most of the time and for most of the cases db can do the write operation with holding only single lock at a time. This enhances the possibility of multiple threads working on different pages at a time and completing their write tasks. The read operation for the Btree again takes single page lock at any given time

The hash implementation is again variant of Extendable hash. The write operation takes single page lock at any given time and read operation doesn't take any lock as it works on validation than locking and finding the record. This increases the read speed for Ehash significantly, in fact read in Ehash is around 50% faster compared to Btree read.

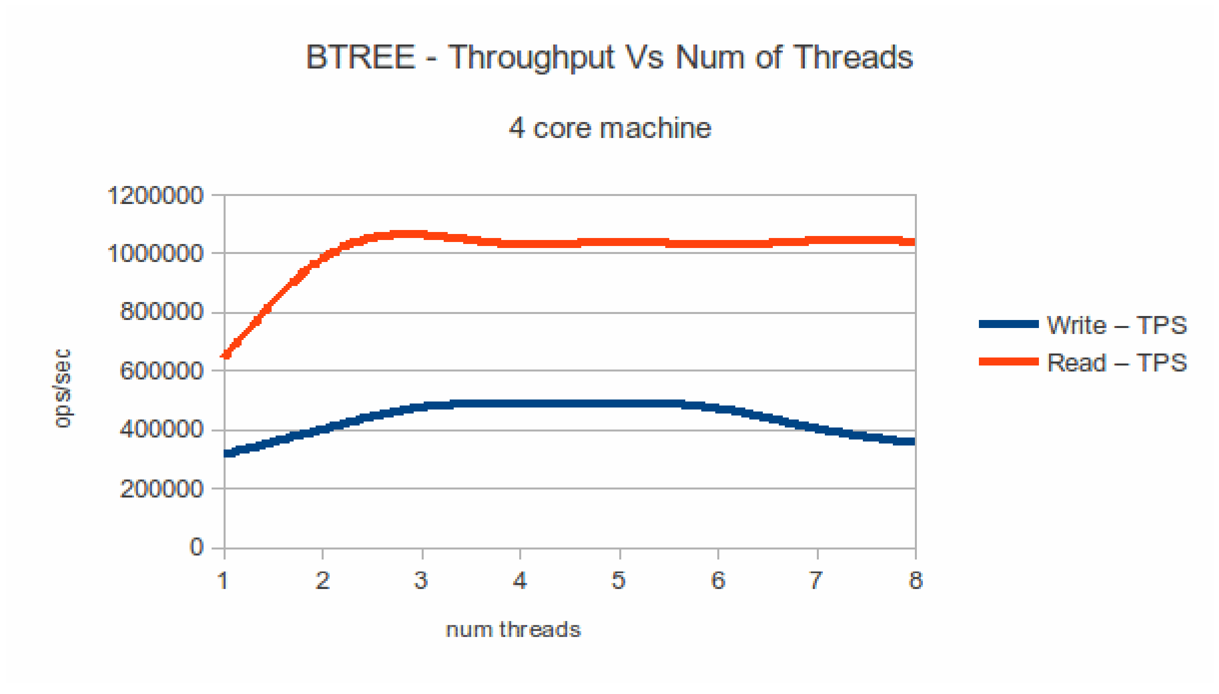
- The buffer pool is highly concurrent

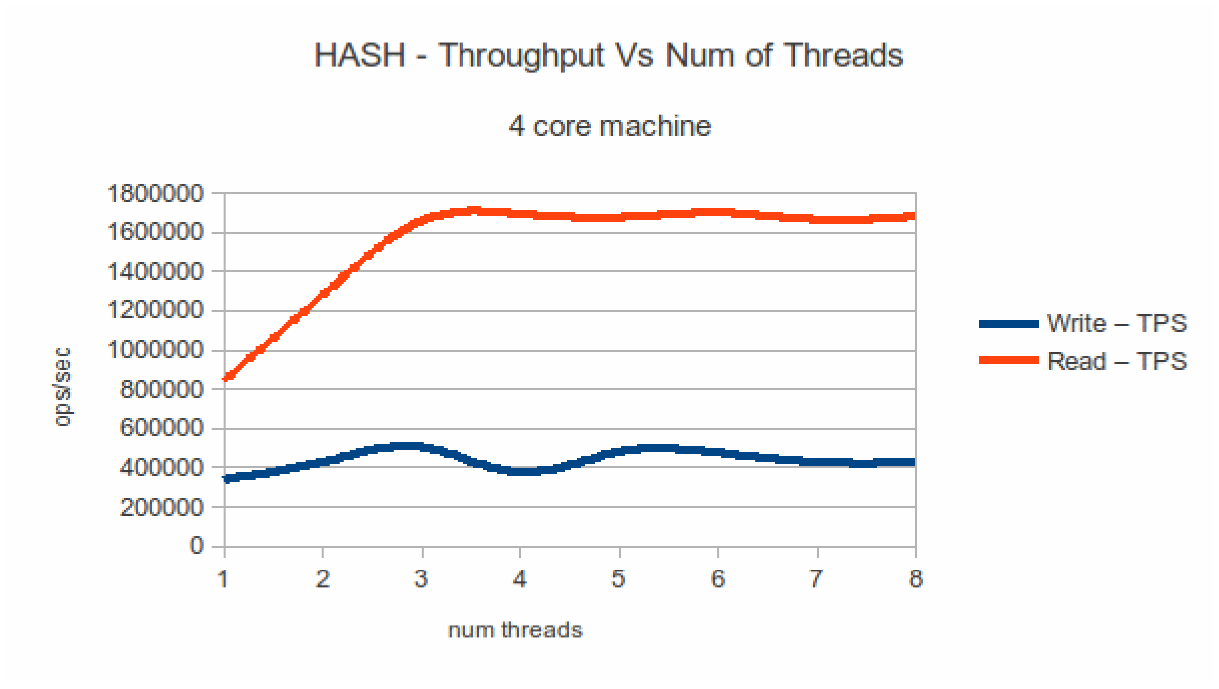
The BangDB creates separate pool for different file types. This is mainly because db maintains different files for index, data, dir etc... and hence keeping separate pools encourages more concurrency for working on the buffer area. For example, to get a buffer header for dat file, whole hash table doesn't need to be locked when a page header for index is required. This boosts multiple threads to work on different pools at any given time hence by increasing concurrency

We can further fine tune the flushing of dirty pages or reclaiming free pages depending upon the nature and criticality of the data. For ex; for Ehash type, dir file size is quite low hence entire data can always be kept in memory without allowing flushing of dirty pages or reclaiming. The free list however is common and it indicates that all has equal rights to get free page when in pressure

- The log is highly concurrent

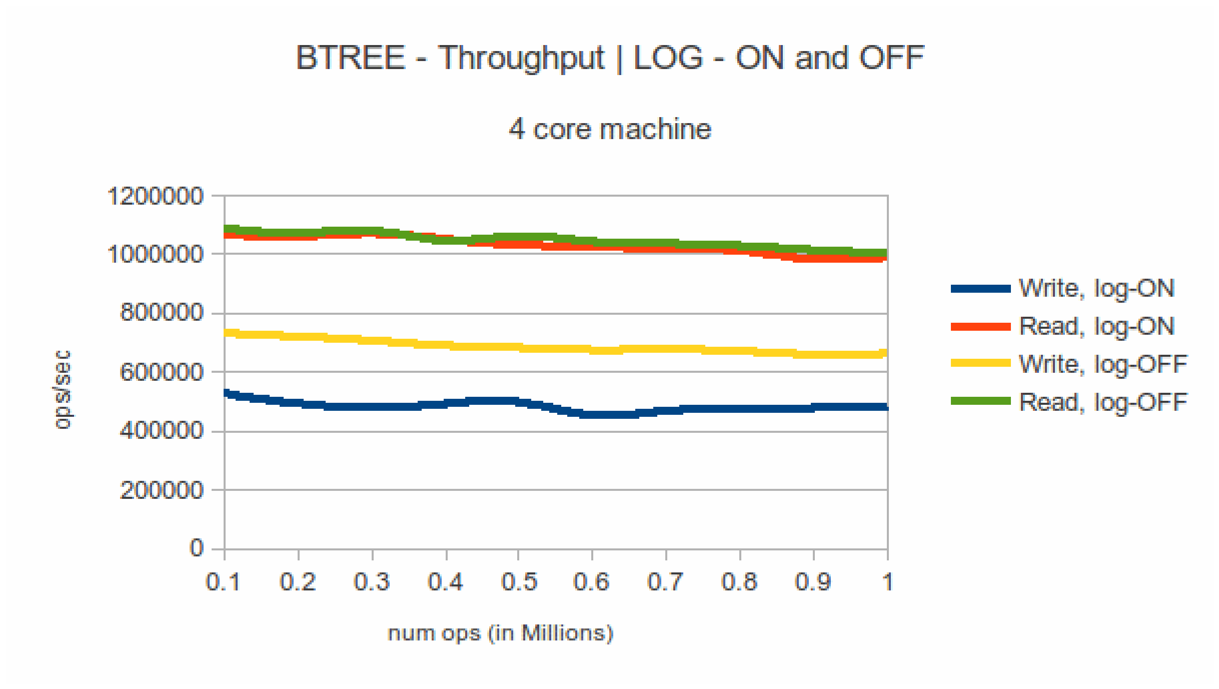
The write ahead log allows multiple threads to write on the log file at a time. Also the various data structures it maintains are also separated with keeping sub use cases in mind. For example; transaction table and dirty page table would require to lock on single header at a time and since thread would already have locked the corresponding page, hence there is no need to lock the writing for that page in the log. Since log is sequential hence flush of log would not require locking the in memory log. Similarly check pointing is also lock free. Only during split of log, the lock is required



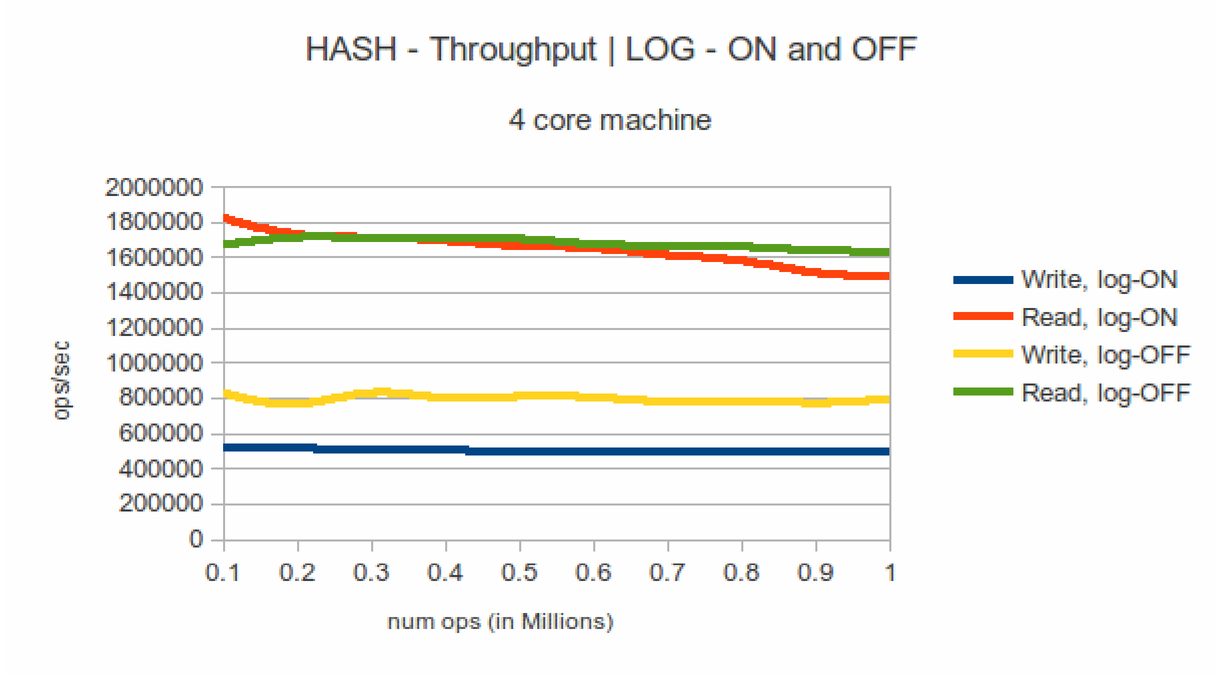


Write Ahead Log

BangDB implements write ahead logging to make the database robust. The db write each and every write operation to the log and updates the data in the buffer pool only. On one hand this enables db to avoid going to disk for all it's operations and deal with the data in RAM. But on the other hand it adds the overhead of writing to the log file and then for durability flush the log to the disk on continuous/continual basis. Hence this section is to analyze the effect of write ahead log on the performance of BangDB



Note that for log switch off, since there is no write to log file in 'read' case, hence we see the performance equal irrespective of whether log is on or off



Btree and Ehash

This section we will cover how Btree and Ehash fares in similar conditions. However few important points to note here as one may want to ponder when to pick which one, and performance certainly, among others play important role.

Btree

Btree arranges the keys in order, default or defined by the user. Hence when we deal with data which has some order and can store them using the same then Btree is a good option. Because Btree has to maintain the order while doing the inserts or updates, it deals with some overhead compared to hash where order is not maintained. Typical complexity for Btree is in the order of logN

Since we are interested in order of the key, hence we may want to retrieve a range of adjacent keys and their values. Btree supports the functionality to do so, whereas hash would not be able to do that. Hence to have scan on range of keys, pick Btree as access method

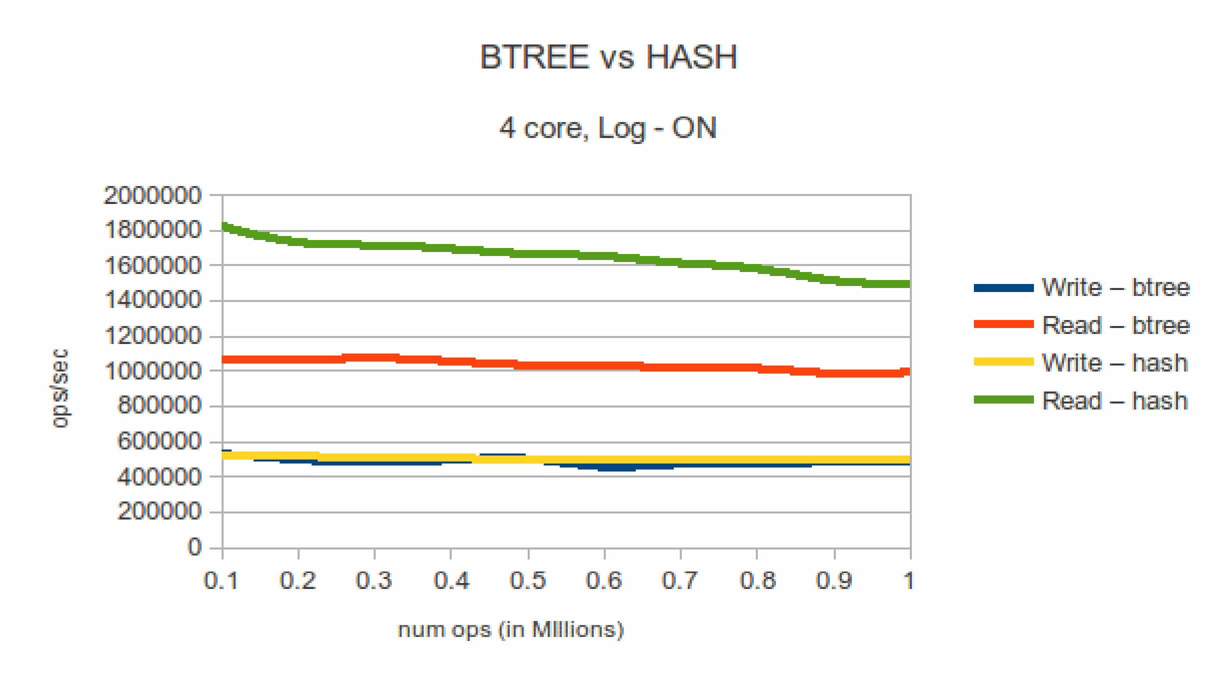
Btree because of it's inherent structure, has balanced read and write, though we will see read performance is better for BangDB than write

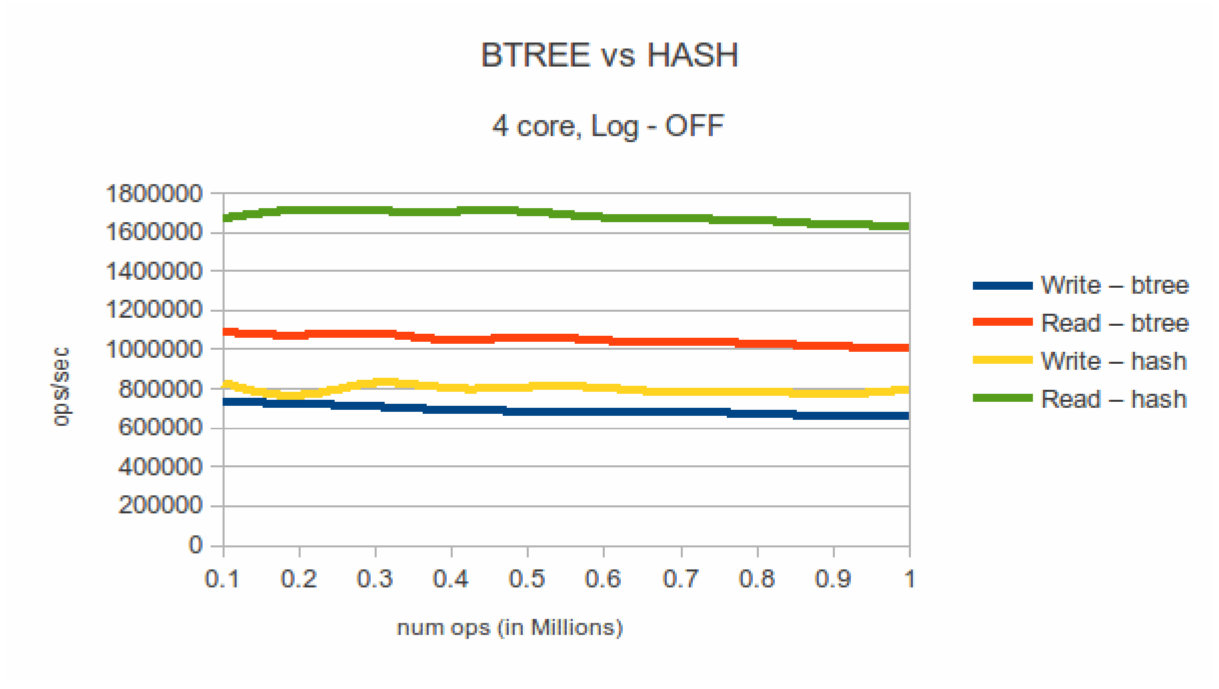
Ehash

Hash stores key in such a manner which guarantees constant time for write and read. BangDB implements extendabale hash, which maintains a directory of all the index pages which in turn stores the keys and data information. Sine directory size is very small compared to index and data files, complete directory is kept in-memory all the time. This enables very fast look up for read and write. Especially for read, ehash doesn't lock any page even in concurrent scenario, which further boosts the read performance

Hence when read performance is desired more than write, then Ehash could be a better choice, especially when order of keys are not important. Note that the write of ehash is again comparable to that of Btree

Following are few graphs which compares the Ehash performance to that of Btree. The comparison has been done for both log On and Off.





Stress Test

Though all the test that we have seen so far in previous pages, are also stress test in a way as we write and read continuously and compute the statistics accordingly. The difference here is that we write much more than the available space in the buffer pool.

Writing more than the available buffer in RAM means that the db has to go to disk for reading and writing data. In non-continuous scenario, there will be time for db to do some flushing when load is less, but in continuous operations, db will have to carry out the space management by flushing dirty pages and reclaiming free pages while operations are going on

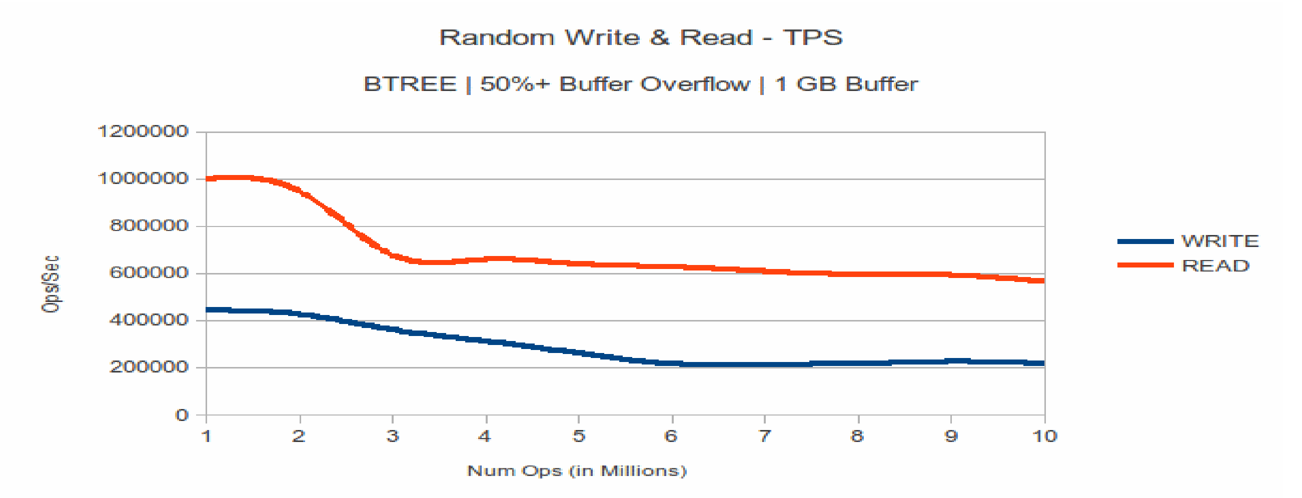
The operations are also totally random, which means it makes the life of db even more difficult, as it can not predict which set of pages to flush and which ones to keep in memory and not free. BangDB implements temporal locality concept to manage space in the buffer pool, which means least recently used will be flushed out before others. This has an implicit assumption (and valid one) that least recently used are least likely to be used in future. But with total random operations this assumption suffers hugely

The BangDB had to do more than just flushing the pages when in crisis. It had to oversee the entire buffer management all the time and start with flushing and reclaiming process much earlier than when it had no other option. This requires a balancing act of not flushing too many as it may result in writing same page multiple times in short span of time. Or not reclaiming too much in advance which will cause cache miss. But at the same be ready with sufficient number of pages in the free list for any operational need. Also keeping directory and index pages more could help as we juggle with these pages to find the data. Also the goal has to avoid the crisis situation where no free page is available in the buffer pool, which will cause longer pause in the system

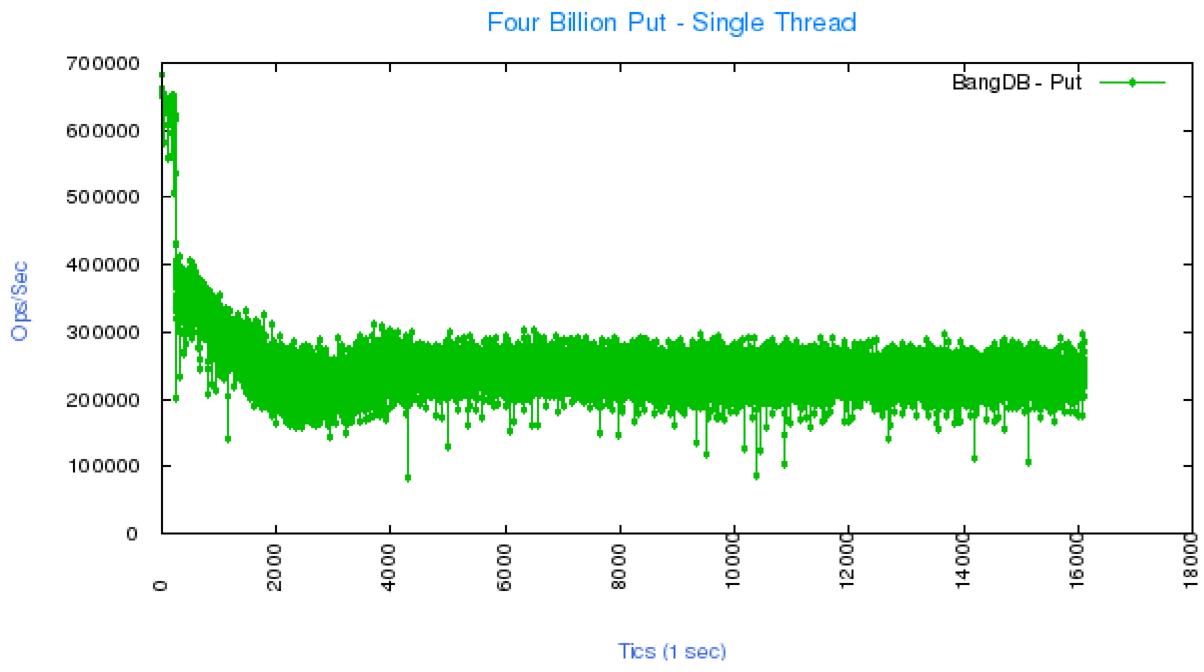
BangDB takes semi adaptive approach towards flushing of dirty pages and reclaiming for free pages in the buffer pool. It's not 100 percent adaptive but something that we should work in future. However, it works pretty nicely when semi sequential or semi random keys are used for Btree. But for total random operations it degrades gracefully and doesn't take longer pauses as experienced with other dbs

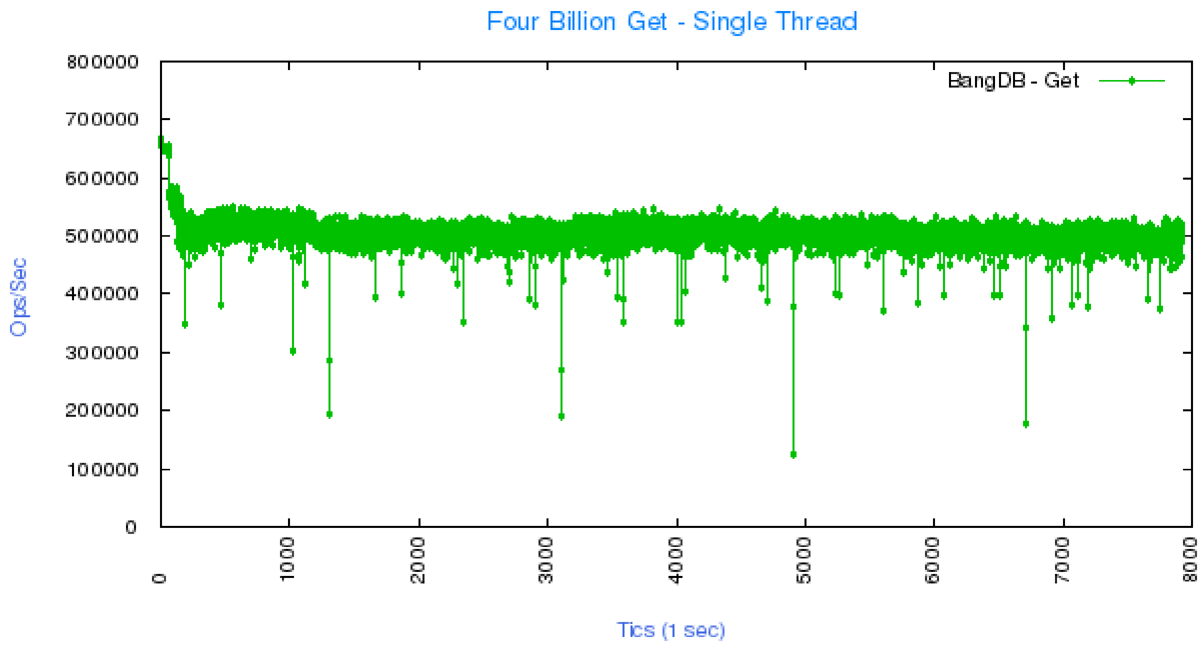
Here is the statistics of writing ~2.25GB of data using 1GB of buffer pool with continuous random writes and reads operations. Note that the db has to flush minimum 1.25GB of data as it can only keep maximum 1GB at any given time. This means that while db was serving the requests (4 threads), it was also freeing up memory in advance for those requests to complete in optimum manner. The log is on hence db is also writing the log file to disk in every 50ms. This is one of worst situations that db will have to face as we don't give db any time for flushing or reclaiming buffer while serving the requests

The numbers shown here are average of few runs for better representation;



The db can be stressed further and run for few hours to see the results. Here is the result of running the db for 4 Billion ops run with single thread on similar configuration machine. This wrote more than 300GB data


Four Billion Put - Single Thread
BangDB - Put
Ops/Sec
Tics (1 sec)


Four Billion Get - Single Thread
BangDB - Get
Ops/Sec
Tics (1 sec)