# BangDB – *Datasheet*

www.iqlect.com

## Overview

In this document, various data points would be discussed around BangDB Embeddable version. The goal of the document is to quickly give some insights from various angle into BangDB to help user to figure out different aspects of the db. We will cover data-points for following items in the document;

- Technical
- Configuration
- Performance

## Technical

Here the highlights of the BangDB from the technical angle are presented. The reason to present these data points is to provide the technical insight in to BangDB and at the same time help user in setting the right expectations from the db;

*The binary*

- The db is implemented in C and C++, but it doesn't use STL or other library
- The db uses only standard headers hence dependency on the OS or platform is minimal
- The BangDB is available as library (dynamic and static) for application to link and use
- The size of binary (library) is around 1MB
- Currently the binary is available in C++, C# and Java on Windows and Linux

*The License*

- BangDB community edition is available free of cost under BSD 3 clause license
- Initially only BangDB library will be released along with headers. Source code will be made public later on
- The core BangDB (embedded and network) will be available free of cost under BSD 3 clause license for community editions only
- There will be licensed version for people who need support and other benefits
- Please contact us at info@iqlect.com to know more on license terms and conditions

*The index, access method*

- The db implements index in two ways at the moment, extend-able hash and B+link Tree
- The index work on single or multiple pages at a time, hence all locking is done at the page level
- The access to index is highly concurrent
- Typical fill ratio for a page is around 55% in the long run, but this can be tweaked using config parameters in different scenario
- The default buffer page size is 8192 but can be configured to other suitable values, again there is no theoretical limit on this
- The limit on key size and data size should be defined upfront by the user. Note that user should decide the key size bit cautiously as it affects the overall performance.

However, data size can be defined bit liberally as it doesn't need to be accurate

- Typically key size should be small compared to the data size, default value for key size is 24 bytes and for data it's 64KB
- Size of the key and the page size dictates the number of keys in a page
- The index page should typically have 32 – 256 keys for better performance, but there is no theoretical restrictions
- On an average, less than two pages are locked while insert using btree and only single lock for read. The ext hash read is completely lock free, hence very fast in concurrent mode

*The buffer pool and page cache*

- The BangDB uses buffer pool and page cache for reading writing data, always when enabled
- When page is found in the cache (cache hit) then its read and written directly. But when it doesn't find the page in the cache(cache miss), then it schedules a read from the disk and then returns the header to the index
- The access to buffer pool headers are concurrent and consistent
- Buffer pool maintains hash table for page headers, lru list for header, dirty page list, and the free hdrs
- The buffer pool and page cache size should be allocated based on parameters like available memory, size of data to operate on, performance need etc... But more the memory the better is performance
- Page cache is strategically partially warmed when started when db is started (if db files are existing) for better performance
- Cache eviction happens to make room for more relevant pages. The cache eviction happens for BangDB by selecting the clean pages and replacing it with other pages. If insufficient clean pages are found then some dirty pages are flushed to make more clean pages available
- To decide what to evict, BangDB uses temporal locality philosophy
- The workers for buffer pool ensures that the dirty pages are written to disk asynchronously, pages are getting freed as needed and also
- The flusher threads writes pages to disk to free memory when memory shrinks below a specified threshold as only clean pages are reclaimed
- The flusher threads also writes the headers which have grown older than some specified threshold, this ensures that the dirty data don't remain dirty forever
- BangDB uses adaptive dirty page flushing methodology which always looks for the moment when flushing of pages could be beneficial
- Multiple sequential buffer pages are clubbed together to reduce the system calls when writing for performance
- In future, BangDB will also support growing and shrinking of buffer dynamically based on need to handle serious high pressure scenario
- When writing to hard disk(typical spindle version which has seek latency), buffer pool reduces the random writes and tries to sequentially write to a file at a time by sorting and clubbing adjacent pages
- The db also uses semi adaptive page pre-fetch algorithm for better performance. It computes the potential future accessed pages and piggy backs the current page fetch as requested by the db. This reduces the disk IO, improves the cache hit and also load

bulk of pages which are mostly sequential

### *The write ahead log*

- The db uses write ahead log for data durability and recovery
- The buffer pool and write ahead log is available as config parameters, users can enable, disable, set values for configuring these etc.
- The workers for log frequently writes the log data to disk, checks for log file for split if needed and accordingly splits the file
- The workers also do the log check-pointing at regular intervals
- Write ahead log works on sequential but circular in-memory log which gets written to disk sequentially
- BangDB extends ARIES write ahead log algorithm for better performance
- BangDB logs both actual data, headers, partial pages or logical operation as required in different cases for better performance and less volume of information to the log file to check unnecessary bloating
- The overhead of log is minimal, but the compression of the log files are left to the users at the moment, this functionality will be added in the next version
- Log file splits when reaches a certain threshold and continues with next sequential log file
- The log memory should also be sufficiently allocated as very less log memory would affect the performance adversely. The current default of 64 MB works fine in most of the high volume and traffic use cases
- The log is always written to disk before data is written for data consistency and durability
- The wal maintains dirty page table and transaction table for various activities
- Few other important features of wal for data recovery are; analyze, redo and undo. They together ensure that as long as log is available data would not be lost for the db
- User can disable or enable log using config handle. When BangDB used just as in-memory volatile cache, log should be disabled as it improves the performance further to great extent

### *Transaction*

- The db implements optimistic concurrency control for enabling the multi ops transactions with full ACID
- The parallel serialization ensures concurrent transactions leveraging the multi cores on the machine
- Transaction is offered to user as configurable parameter, hence same db can be opened in different modes by the user
- Autocommit is provided as another configurable parameter which user can enable or disable based on need
- WAL supports the undo and crash recovery for the given transactional boundaries as needed

*Miscellaneous*

- Different files for logs, data, index are used for BangDB. Data file has .dat extension, index file has .idx, directory file has .dir (only for ext hash), log file has .log and metadata file has .met extension

## Configuration

There are several configurable parameters for BangDB which set properly can let the db perform in best possible manner in different context and conditions. Here this section tries to give some basic simple guidelines to set few important parameters depending upon condition. Again for detail list of all these configurable parameters please see the relevant document in the resource section on the site.

### *1 persistence type*

The BangDB can be configured to run in different ways and one of the most important parameters to set is the persistence type. The Embeddable BangDB can run in three possible ways;

1.1 INMEM_ONLY: As the name suggests, this applies for Embeddable (“EMBED”) db only and the db works completely out of the memory. This means the db will never go to disk or other media for any data requirements even though the buffer might be reaching the threshold. Hence there is no overflow of data to disk in case the buffer is full. It works totally as in-memory non persistent cache. User needs to allocate enough buffer space for the db to work properly. When buffer gets full, the db will start throwing errors telling that the buffer is full and reject all write operations. Hence this option should be used when user is sure of not exceeding the buffer amount.
Since persistent is not required in the case, hence ongoing logging should be disabled as well. The db performance is best in this case as there is no logging, flushing of data, read from disk etc. In the end user may want to take the snapshot of the data and persist it, hence to do that user may just call dumpdata() to store the data on the disk before closing. Single process can access the db at a time but multiple threads but with multiple threads

1.2 INMEM_PERSIST: This is for running the db as persistent store. Here the db will overflow the data from the buffer pool to the disk as and when required. The write ahead logging should be enabled if needed as it may ensure data durability and recoverability, but in case data is not too critical then logging may be disabled too. Note that BangDB still works out of buffer pool, but as and when required it flushes dirty pages, reclaim free pages, reads data in case of cache miss etc. to ensure that all operations are done transparently to the user irrespective of the kind of IO that it might be doing to complete the operation. Single process can access the db at a time but multiple threads but with multiple threads

1.3 PERSIST_ONLY: This is for a small set of scenarios where user would like BangDB to work similar to classical dbm fashion. In this case BangDB writes and reads data from the file on the disk. There is no buffer pool and logging for this approach. This

is useful for a scenario where data volume is low but it's highly critical and has to be persisted as soon as it's written. Multiple process can access the database but in single threaded fashion in this case

2 ***index type***

2.1 *EXTHASH* - This is for extend-able hash based index support. This doesn't maintain the order of the keys in the index files but uses hash value of the key to read and write the data. Hence the scan or range scan can't be used in this index type. The constant look up time for keys in this case provides the maximum performance for the read operation. Please see the perf doc for stats

2.2 *BTREE* - This is for B+ link Tree based index support. This maintains the order of the keys while write and hence the range scan is possible in this case. This performs equally good in both read and write. Please see the perf doc for stats

3 **log** - BangDB implements write ahead log for data recovery and durability. But user has an option to enable or disable the log depending upon need. For example for EMBED_INMEM_ONLY db type user may want to disable log as application may not be interested in the persisting the data on the disk. Disabling log further improves the performance significantly.

4 **page size** - Default page size used by db is 8192 bytes. But user can change it to any size that he/she thinks fit. Note that having too high or too low page size has it's own pros and cons, but maintaining a balance is important for getting optimal performance. The various factors that could lead to determination of page size could be, (a) avg size of keys (b) avg size of data (c) single threaded vs multi-threaded etc.

5 **data size** - This denotes the maximum size of data supported by the db. Default set value is 1 MB but the user can change it to higher or lower value as suited. Note that this value has some impact on the available buffer pool size and RAM size of the machine. If we have higher RAM on the machine, then we can commit more memory to the buffer pool hence typically can handle higher sizes of the data. For moderate 256MB buffer pool. We can live with 1-4MB of the maximum data sizes, and if we have 1GB for buffer pool we can go for 1-16MB as well. Note that there is no theoretical limit on this but for practical purposes we should not over commit this value. For data beyond the set maximum size, user will have to split and write and the get and combine accordingly

6 **index size/key size** - Default size for index is 32 bytes (the key size). This again has no theoretical limit but one should be careful in setting this. Too high value would lead to reduced performance for both read and write. The key size has to be less than the page size, typically user should provide the space for at-least 16 keys per page, but more is better. In all probabilities the key sizes would remain limited and in few bytes only hence set this according to the requirement without bloating index size

7 **buffer pool size** - This sets the buffer pool size. This is again hint to the db and BangDB then computes the best possible value for buffer pool less than but as close to this hint as possible. Providing higher value is always good but it depends on the machine configuration as well

Please see the configuration doc for detail information on this.
Also please see other resources available on BangDB at the site for get the detail insight of the db.

## Performance

Here the IOPS summary from the high level are presented in various conditions. For the hardware and software configuration please see the Benchmark document for details.

There are many scenarios in which the IOPS for the db could be measured. We have measured the value to help users in few such scenarios and from there user can understand what to expect from BangDB as far as IOPS is concerned.

The average values for the throughput(ops/sec) over the 1M – 10M operations

### Scenario 1: All operations from the buffer pool, log = OFF

In this scenario, we write and read completely from the buffer pool with out going into the hard disk. Log is off in this case. The other configuration are same as described in the Benchmark page in the section. This maps to the real world scenario of working out as cache where log in seldom needed

| Index | Write (ops/sec) | Read (Ops/sec) |
|---|---|---|
| Btree | 685,000 | 1,045,000 |
| Ehash | 790,000 | 1,675,000 |

### Scenario 2: All operations from the buffer pool, log = ON

In this scenario, we write and read completely from the buffer pool with out going into the hard disk. Log is on in this case. The other configuration are same as described in the Benchmark page in the section. This maps to the real world scenario where cache is in-memory but would like to take snapshot of the log for future purposes

| Index | Write (ops/sec) | Read (Ops/sec) |
|---|---|---|
| Btree | 475,000 | 1,025,000 |
| Ehash | 500,000 | 1,690,000 |

### Scenario 3: Operations involving hard disk access some of the time, log = OFF

In this scenario, we write and read mostly from the buffer pool(wrote 20% more than the buffer pool capacity) with going out to the hard disk whenever data was not present in the bufferpool. Since we write and read random data hence db has to go to hard disk more often than not. Log is off in this case. The other configuration are same as described in the Benchmark page in the section. This maps to the real world scenario for corner cases of store being persistent but the data durability is of less importance compared to sheer speed of the operation

| Index | Write (ops/sec) | Read (Ops/sec) |
|---|---|---|

| | | |
|---|---|---|
| Btree | 570,000 | 830,000 |
| Ehash | 625,000 | 1,275,000 |

### Scenario 4: Operations involving hard disk access some of the time, log = ON

In this scenario, we write and read mostly from the buffer pool(wrote 20% more than the buffer pool capacity) with going out to the hard disk whenever data was not present in the bufferpool. Since we write and read random data hence db has to go to hard disk more often than not. Log is on in this case. The other configuration are same as described in the Benchmark page in the section. This maps to the real world scenario for most of the cases of store being persistent and the data durability is of high importance. But we generally don't write and read to db to the maximum of it's capacity continuously all the time hence real world would treat db much better than this

| Index | Write (ops/sec) | Read (Ops/sec) |
|---|---|---|
| Btree | 355,000 | 815,000 |
| Ehash | 365,000 | 1,300,000 |

### Scenario 5: Operations involving hard disk access most of the time, log = ON

In this scenario, we write and read less from the buffer pool(wrote 100% more than the buffer pool capacity) with going out to the hard disk most of the time when data was not present in the bufferpool. Log is on in this case. The other configuration are same as described in the Benchmark page in the section. This is an extreme scenario where we continuously write much more than the buffer pool can handle. Db consistently flushes out data and reads data into the pool, read write operations happen in parallel

| Index | Write (ops/sec) | Read (Ops/sec) |
|---|---|---|
| Btree | 200,000 | 600,000 |
| Ehash | 190,000 | 820,000 |