BangDB – *Getting Started*

www.iqlect.com

Installation

The Bangdb comes as library (both static and dynamic) to be installed on the machine where user would like to run applications which will use embeddable Bangdb. The installation of Bangdb is simple and clean. The installer would just copy the library in /usr/local/lib folder, it will also create the soft links as required. The installer would then copy the headers in the /usr/local/include/bangdb/ dir. It will also create a bangdb group on the machine and add the user to it. The UN-installer would simply remove the library and the soft links, the header files and delete the bangdb group. Hence the UN-installation is pretty clean in that sense too.

To get the latest version of the library, please download it from www.iqlect.com. Please see the README for relevant information on the product.

To get started, here is a sample code for simple operations. Note that to use Bangdb from the applications, please include database.h file.

```
#include <bangdb/database.h>
int main()
{
	//create database, table and connection
	database* db = new database((char*)"userdb");

	//get a table, a new one or existing one
	table* tbl = db->gettable((char*)"userinfo");
	if(tbl == NULL)
	{
		bangdb_logger("ERROR:table NULL error");
		exit(-1);
	}

	//get a new connection
	connection* conn = tbl->getconnection();
	if(conn == NULL)
	{
		bangdb_logger("ERROR:connection NULL error");
		exit(-1);
	}

	bangdb_logger("db, table and connection created\n");

	//data type to be used
	FDT *fk, *fv, *fout;

	const char *key = "my_key_", *val = "my val", *val2 = "my new val";

	//create FDT type with key and val
	fk = new FDT((void*)key, strlen(key));
	fv = new FDT((void*)val, strlen(val));

	int retval = 0;

	//now insert
	if((retval = conn->put(fk, fv, INSERT_UNIQUE)) < 0)
		bangdb_logger("error in insert\n");
	else if(retval == 1)
		bangdb_logger("key already exists\n");
	else
		bangdb_logger("insert successful");


	//get the val
	if((fout = conn->get(fk)) == NULL)
		bangdb_logger("error in get\n");
	else if(strncmp((char*)fout->data, val, strlen(val) != 0))
		bangdb_logger("mismatch in get\n");
	else
		bangdb_logger("get successful");
```

```
	//release fout
	if(fout)
		fout->free();
	delete fout;


	//update key
	delete fv;
	fv = new FDT((void*)val2, strlen(val2));

	if((retval = conn->put(fk, fv, UPDATE_EXISTING)) < 0)
		bangdb_logger("error in update\n");
	else
		bangdb_logger("update successful");

	//delete key
	if((retval = conn->del(fk)) < 0)
		bangdb_logger("error in delete\n");
	else
		bangdb_logger("delete successful");

	bangdb_logger("starting clean up");

	//clean up
	delete fv;
	delete fk;

	conn->closeconnection();
	delete conn;
	tbl->closetable();
	delete tbl;
	db->closedatabase();
	delete db;

	bangdb_logger("test done");
	return 0;
}
```

NOTE: we can do all for char* type as well instead of FDT if we want to, APIs are same but they take char* instead of FDT*. Also in case of char* since we don't create FDT type hence no clean up is needed for keys and values, but in case of FDT clean up is the responsibility of the developer

Configuring Bangdb

The Bangdb provides a range of configurable parameters to run the db in a particular way most suitable for the application. These parameters dictates the way db will run, store data and the degree of durability as well apart from many other elements like performance etc. To see the complete list of these parameters and their meaning, please see the bangdb configuration document, but here some important ones are listed to get started;

1 *persistent type*

The Bangdb can be configured to run in different ways and one of the most important parameters to set the run configuration is the db 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.