BangDB – API (Embedded Version 1.5)

CPP

www.iqlect.com

The document covers the API for Bangdb, embeddable version. The API for C++ is covered here.

API for C++

database

1. To create database;

```
database(char *dbName, char *configPath = NULL, db_transaction_type tranType = DB_TRANSACTION_NONE,
char *dbpath = NULL);
```

The tranType by default is *DB_TRANSACTION_NONE* which means db is not opened in transaction mode and dbpath by default is NULL which means read the dbpath(where the db files will be stored, SERVER_DIR in the bangdb.config file) from the config file. If you wish to override the path provided in the bangdb.config file then pass the path value here.

ConfigPath is also by default NULL which means find the bangdb.config in the local project dir, else provide the full dir path to it (the bangdb.config file).

If successful, db is created and initialized and returned a pointer to database, else program quits with error message

db_transaction_type has following values;

DB_TRANSACTION_NONE,
DB_OPTIMISTIC_TRANSACTION,
DB_PESSIMISTIC_TRANSACTION,

As of now the DB_PESSIMISTIC_TRANSACTION type is not supported

2. To get a table;

```
table* gettable(char *name, bangdb_open_type openflag = OPENCREATE, table_env *tenv=NULL);

wideTable* getWideTable(char *name, bangdb_open_type openflag = OPENCREATE, table_env *tenv=NULL);

table* getPrimitiveTable(char *name, bangdb_primitive_data_type dataType = PRIMITIVE_LONG,
bangdb_open_type openflag = OPENCREATE, table_env *tenv=NULL);
```

As you notice there are three APIs to create three different types of tables, please see the documentation http://iqlect.com/developer.php to get more info on different table types

The bangdb_open_type has following values;

OPENCREATE,
TRUNCOPEN,
JUSTOPEN

Most of the time OPENCREATE is the right type to use as it creates the db files if they don't exist else just opens them

The table_env is a type which can be created, set with proper values and passed to the gettable. This allows user to override a set of config values as defined in the config file. Note that if NULL is passed for table_env then db config (defined in bangdb.config) values will apply to the table as well, but user can override the values by creating table_env for each table. This allows us to treat each table differently. Following parameters can be set or overridden for the table;

bangdb_persist_type

INMEM_ONLY,
INMEM_PERSIST,
PERSIST_ONLY,

bangdb_index_type

HASH, - deprecated
EXTHASH,
BTREE,
HEAP – Not supported

key size

size of key in bytes, user should select this properly as lower the value better the db would perform. Also once defined for a table this value can't be changed later

key type

NORMAL_KEY,
COMPOSITE_KEY,
NORMAL_KEY_LONG

allow duplicates (boolean)

true,
false
This tells if duplicates are allowed for keys or not. Default is false

log (wal) state

true,
false,

bangdb_log_type (private or shared log)

SHARED_LOG,
PRIVATE_LOG,

log size in MB

size of log file in MB

auto commit

true,
false,

table type (default is NORMAL_TABLE)

NORMAL_TABLE,
WIDE_TABLE,
INDEX_TABLE, //not to be used by developers, only for BangDB
PRIMITIVE_TABLE_INT,
PRIMITIVE_TABLE_LONG,
PRIMITIVE_TABLE_STRING,

Typically we don't set these values in the table_env rather call the appropriate API to create NORMAL, WIDE or PRIMITIVE table as defined above (the API)

table size hint (This will be discussed separately, default is good for most cases)

TINY_TABLE_SIZE,
SMALL_TABLE_SIZE,

NORMAL_TABLE_SIZE,
BIG_TABLE_SIZE,

sort method

LEXICOGRAPH,
QUASI_LEXICOGRAPH

sort direction

SORT_ASCENDING,
SORT_DESCENDING

Hence table_env gives us the capability to create multiple tables withing a db with different set of config values. For example, we can create table1 with {inmem_only, hash, log-on..}, table2 with {inmem_persist, Btree, log-off, autocommit-on...} and so on

It is good practice to use table_env for each table otherwise the default values (as defined in bangdb.config file) would be used for all the tables.

When successful, this returns a pointer to the table object else NULL with error message

3. Close database

void **closedatabase**(bangdb_close_type dbclose = *DEFAULT*);

The possible values for bangdb_close_type are;

DEFAULT,
CONSERVATIVE,
OPTIMISTIC,
CLEANCLOSE,

Note that closedatabase is enough to close the database and user need not close each table or connection separately. Closing database closes everything and cleans up all resources. But it is recommended to close all connections and tables as well and delete the objects before closing the database

4. Close table

int **closetable**(char *tblName, bangdb_close_type tblclose = *DEFAULT*);

int **closetable**(table *tbl, bangdb_close_type tblclose = *DEFAULT*);

int **closetable**(wideTable *tbl, bangdb_close_type tblclose = *DEFAULT*);

Returns 0 on success and -1 for error with message
It is recommended to use the closetable() API exposed by the table itself as that's more intuitive

5. Beginning a transaction

void* **begin_transaction**();

Returns a opaque pointer which should be used in all operations participating in the transaction

6. Committing a transaction

FILEOFF_T **commit_transaction**(void **txn_handle);

When successful, returns a unique transaction id else -1 on error with message

7. Aborting a transaction

void **abort_transaction**(void **txn_handle);

table

1. Getting a connection

connection* **getconnection**();

primConnection* **getPrimConnection**();

Returns a pointer to connection object else NULL with error message

Note that for NORMAL_TABLE user should call getconnection() and for PRIMITIVE TABLE (all types of primitive tables) user should call getPrimConnection. If there is mismatch in calling either of the APIs then it returns NULL with proper error message.

2. Closing this table

int **closetable**(bangdb_close_type tableclose = *DEFAULT*);

Returns 0 for successful close else -1 with error message

3. Closing connection

```
int closeconnection(connection *conn);
```

Return 0 for successful close else -1 with error message
It is recommended to use the closeconnection() API exposed by the connection itself as that's more intuitive

4. Dumping the data

int **dumpdata**();

This is helpful in explicitly dumping the data to disk at any time during the operation. But mostly used when we run the table in inmemory only mode but later during close or any other time we wish to save the data on disk

Returns 0 for successful close else -1 with error message

wideTable

1. Getting a connection

wideConnection* **getconnection**();

Returns a pointer to connection object else NULL with error message

2. Closing this table

int **closetable**(bangdb_close_type tableclose = *DEFAULT*);

Returns 0 for successful close else -1 with error message

3. Adding/Creating Index

```
int addIndex_str(char *idx_name, int idx_size, bool allowDuplicates);
```

```
int addIndex_num(char *idx_name, bool allowDuplicates);

int addIndex(char *idx_name, table_env *tenv);

int dropIndex(char *idx_name);
```

These API are self explanatory, they create/add and drop index for the given field for the table. The flag allowDuplicates tells if the duplicates are allowed or not.

The addIndex_str creates index for key which will be treated as string or opaque, the addIndex_num is for creating index for key which will be treated as number. The addIndex is generic one and user can set various config values using table_env

4. Dumping the data

int **dumpdata**();

This is helpful in explicitly dumping the data to disk at any time during the operation. But mostly used when we run the table in in-memory only mode but later during close or any other time we wish to save the data on disk
Returns 0 for successful close else -1 with error message

5. Closing all active connections

```
int closeAllConnections();
```

connection (For Normal Table)

1. Putting key value without transaction

FILEOFF_T **put**(char *key, char *val, insert_options flag);

FILEOFF_T **put**(FDT *key, FDT *val, insert_options flag);

```
FILEOFF_T put(char *key, int keylen, DATA_VAR *val, insert_options flag);

FILEOFF_T put(LONG_T key, FDT *val, insert_options flag);

FILEOFF_T put(LONG_T key, char *val, insert_options flag);
```

FILEOFF_T is long long type

char* key and values are array of characters and FDT is a type with following structure;

```
struct FDT
{
        DATLEN_T length;  //DATLEN_T is unsigned int
        void *data;               //opaque piece of data
};
```

DATA_VAR type is used when user wishes to use pre-allocated buffer. It also allows user to define offset within buffer from which read or write should be done. The db sets the appropriate error/success flag as well after the operations. The DATA_VAR is defined as follows;

```
struct DATA_VAR
{
        void *data_buf;         //the pointer to pre allocated buffer. for write db will read from
                                //it and for read db would write to it. User should clear the buffer
        int data_buf_len;       //the allocated buffer length (db doesn't touch this)
        int data_len;           //length of the read data (returned by db) or length of data to write
        int data_offt;          //for write db takes the bytes from data_buf from the data_offt and
                                //for read db reads data in db from data_offt
                                //data_offt + data_len <= data_buf_len, else data will be  curtailed
        int flag;               //db sets if more data is to come for read(MORE_DATA_TO_READ)
}
```

The insert_options has following options;

INSERT_UNIQUE,
UPDATE_EXISTING,
INSERT_UPDATE,
DELETE_EXISTING,

Return 0 for success else -1 with error message

2. Putting key value with transaction

FILEOFF_T **put**(char *key, char *val, insert_options flag, void *txn_handle);

FILEOFF_T **put**(FDT *key, FDT *val, insert_options flag, void *txn_handle);

```
FILEOFF_T put(char *key, int keylen, DATA_VAR *val, insert_options flag, void *txn_handle);
```

These are similar to above APIs except that they take transaction handle

Return 0 for success else -1 with error message

3. Getting value for a key without transaction

char* **get**(char *key);

FDT* **get**(FDT *key);

```
int get(char *key, int keylen, char **val, int *vallen);
```

```
int get(char *key, int keylen, DATA_VAR *data);
```

```
FDT *get_data(LONG_T key);
```

returns appropriate values or NULL with error message

4. Getting value for a key with transaction

char* **get**(char *key, void *txn_handle);

FDT* **get**(FDT *key, void *txn_handle);

```
int get(char *key, int keylen, char **val, int *vallen, void *txn_handle);
```

These are similar to above APIs except that they take transaction handle

Returns appropriate values or NULL with error message

5. Deleting a key value without transaction

FILEOFF_T **del**(char *key);

FILEOFF_T **del**(FDT *key);

```
FILEOFF_T del(LONG_T key);
```

```
FILEOFF_T del(int key);
```

Return 0 for success else -1 with error message

6. Deleting a key value without transaction

```
FILEOFF_T del(char *key, void *txn_handle);

FILEOFF_T del(FDT *key, void *txn_handle);

FILEOFF_T del(LONG_T key, void *txn_handle);
```

Return 0 for success else -1 with error message

7. Range query or scan without transaction

```
resultset* scan(char *skey, char *ekey, scan_filter *sf = NULL);

resultset* scan(FDT *skey, FDT *ekey, scan_filter *sf = NULL);

resultset* scan(LONG_T skey, LONG_T ekey, scan_filter *sf = NULL);

resultset* scan(int skey, int ekey, scan_filter *sf = NULL);
```

Here skey and ekey are start and end keys for range query. Either of these two keys can be NULL, partial, full or some random value keys. When NULL is provided for skey and/or ekey then it means full scan in that direction. For example to get all values for keys greater than equal to key="mykey" we can use scan(key, (FDT*)NULL) or to get get all values (select * from table) we can use scan((FDT*)NULL, (FDT*)NULL)

scan_filter is provided to further add operators while doing the query and provide some limits on the returned resultset. The values for scan_filter is as follows;

```
scan_operator skey_op;    //default GTE
scan_operator ekey_op;    //default LTE
scan_limit_by limitby;    //default LIMIT_RESULT_SIZE
int limit;                //default 2MB (MAX_RESULTSET_SIZE)
```

The scan_operator has following values;

```
GT,     //greater than
GTE,    //greater than equal to - default
LT,     //less than
LTE,    //less than equal to – default
```

And scan_limitby has following values;

```
LIMIT_RESULT_SIZE,    //defines the bytes of data that should be returned (max)
LIMIT_RESULT_ROW,     //number of rows (max) that should be returned
```

The resultset is scrollable cursor with methods to iterate over the returned keys and values based in the given parameters by scan.

Returns resultset on success else NULL for error

8. Range query or scan with transaction

```
resultset* scan(char *skey, char *ekey, void *txn_handle, scan_filter *sf = NULL);

resultset* scan(FDT *skey, FDT *ekey, void *txn_handle, scan_filter *sf = NULL);

resultset* scan(LONG_T skey, LONG_T ekey, void *txn_handle, scan_filter *sf = NULL);
```

These are similar to above APIs except that they take transaction handle.returns appropriate values or NULL with error

message

9. Counting the number of items

```
FILEOFF_T count(char *skey, char *key, scan_filter *sf = NULL);

FILEOFF_T count(FDT *skey, FDT *key, scan_filter *sf = NULL);

FILEOFF_T count();

LONG_T count(LONG_T skey, LONG_T ekey, scan_filter *sf = NULL);

LONG_T count(int skey, int ekey, scan_filter *sf = NULL);
```

This is very similar to scan except that this returns the count instead of resultset. The skey and ekey behavior is similar

Third one is convenient method for counting all elements in the table

Returns count(FILEOFF_T is long long) on success else -1 on error

10. Setting auto commit

```
void set_autocommit(bool flag);
```

When transaction is OFF, then autocommit by default is ON and all individual operations guarantees ACID. We use non transactional APIs for get, put, del, scan. But when transaction is on then if autocommit is off then we can only use transactional APIs for get, put, del, scan and all non-transactional ones will fail. But to override the scenario, one can call this method and perform transactional or non transaction ops as required.

For example, if db is opened in transactional mode with autocommit = OFF then following is not allowed;

```
char *v = conn->get(k);          //Not OK
```

we will have to either use the transactional API,

```
void *txn = db->begin_transaction();
char *v = conn->get(k, txn);
```

Or, set the autocommit ON,

```
conn->set_autocommit(true);
char *v = conn->get(k);          //OK
```

11. Close this connection

```
int closeconnection();
```

Returns 0 for success else -1 with error message

primConnection (for primitive table)

The main difference in the get and put APIs since the value for primitive table will always be either int or long. Also transaction is not supported for primitive table as of now in 1.5. Rest of the other APIs are similar to that of connection.

1. get

```
int get(LONG_T key, LONG_T *val);

int get(int key, int *val);

int get(FDT *key, LONG_T *val);

int get(char *key, LONG_T *val);
```

2. put

FILEOFF_T **put**(LONG_T key, LONG_T val, insert_options flag);

FILEOFF_T **put**(**int** key, **int** val, insert_options flag);

FILEOFF_T **put**(**char** *key, LONG_T val, insert_options flag);

FILEOFF_T **put**(FDT *key, LONG_T val, insert_options flag);

wideConnection (for wide table)

Most of the APIs are similar to that of connection, except that it has few more APIs for put and scan and they are;

1. scan

resultset ***scan**(**char** *index_name, FDT *skey, FDT *ekey, scan_filter *sf = NULL);

resultset ***scan**(**char** *index_name, **char** *skey, **char** *ekey, scan_filter *sf = NULL);

resultset ***scan_doc**(**char** *index_name, FDT *skey, FDT *ekey, scan_filter *sf = NULL);

resultset ***scan_doc**(**char** *index_name, **char** *skey, **char** *ekey, scan_filter *sf = NULL);

resultset ***scan_doc**(**char** *index_name, LONG_T skey, LONG_T ekey, scan_filter *sf = NULL);

2. put

int put(FDT *key, FDT *val, **char*** index_name, FDT *index_val);

int put(LONG_T key, FDT *val, **char*** index_name, FDT *index_val);

int put_doc(**char** *key, **char** *json_str, insert_options flag);

int put_doc(LONG_T key, **char** *json_str, insert_options flag);

LONG_T put_doc(char *json_str);

resultset

1. Checking if the result has next key, val pair

bool **hasNext**();
returns true if yes and false otherwise

2. Moving to the next key and val pair

void **moveNext**();

3. Beginning the iteration

void **begin**();

```
void begin_reverse();
```

When resultset is returned, it is set at beginning, but if we want to go back to beginning then we call this api. Note that if you wish to traverse in reverse direction all you need to do is just call the begin_reverse() before iterating and then use the same APIs for moving next or getting key and value

4. Getting the next key

FDT ***getNextKey**();

```
long getNextKeyLong();
```

```
char * getNextKeyStr();
```

First one returns FDT type for key or NULL, second one is used for int/long data type and third one is for string

5. Getting the next val

FDT ***getNextVal**();

```
long getNextValLong();
```

```
char * getNextValStr();
```

First one returns FDT type for key or NULL, second one is used for int/long data type and third one is for string

6. Last Evaluated Key

FDT ***lastEvaluatedKey**();

```
long lastEvaluatedKeyLong();
```

This returns the last key in the resultset. Please see the documentation for use of this method

returns the FDT type for key

7. Check if more data is to come

bool **moreDataToCome**();

returns true if more data is expected to come for the query else false

8. Counting the num of items in the resultset

int **count**();

returns the number of items in the resultset

9. Searching a key in the resultset

FDT ***searchVal**(FDT *key);

returns the value for the given key

10. Freeing the resultset resource

void **clear**();

11. Operation on two resultset

To allow user to be able to scan table using different ways and the create a new resultset by adding, appending or intersecting the two different resultsets, following APIs are provided;

```
void add_doc(resultset *rs, char *orderBy=NULL);

void add(resultset *rs);

void append(resultset *rs);

void intersect(resultset *rs);
```

Sliding Table

Sliding table is used for storing items only within the sliding window. This is again very useful for real time analysis. This allows user to keep data only for the given sliding window.

SwTable is the type that offers the APIs to deal with sliding window table and the APIs are as following. Note that the APIs are very similar to that of other tables in the database

1. create

   ```
   swTable(database *db, char *tableName, table_env *tenv, int ttlsec, bool archive = false)
   ```

   Creates the sliding window table. The archive option is to tell if you would wish to archive the older data or simply discard them

2. initialize

   ```
   int initialize()
   ```

   The user needs to initialize the sliding window table before it can be used

3. add index

   ```
   void addIndex(char *idxName, table_env *tenv)
   ```

4. getconnection

   ```
   wideConnection *getConnection()
   ```

5. put

   ```
   int put(char *str, int len, insert_options iop)

   int put(char *str, int len, char *idx, char *idxkey)
   ```

6. scan

   ```
   resultset *scan(int period)

   resultset *scan(int period, int lag)
   ```

   The scan is here for a period starting from now upto sometime in the past, for example last 30 min would put period as 30*60. Similarly, user can specify the lag from now for the period as specified

7. close

   ```
   void close()
   ```

Counting

BangDB supports two types of counting;

A) Absolute Counting
B) Probabilistic Counting

For each of the above counting we can further define following attributes;

a) Unique Counting
b) Non-Unique Counting

The BangDB makes the counting feature available through swEntityCount type. Which basically allows user to create a sliding window within which they can count items. Rest of the details of sliding window is handled by the type itself and user won't have to do anything more than calling the API appropriately. The exposed APIs are;

1. create the object of type

```
swEntityCount(char *countName, int swTime, int swExpiry, int nEntity = 32);
```

User will be able to create the sliding window for counting entities by calling the abpve constructor. The `countName` refers to the counting type using which we want to count, swTime is the sliding window time, swExpiry is the time of expirtation of items. The nEntity is the hint to the type on how many entities are going to be counted by the type

2. create entity

```
void createEntity(char *name, bangdb_window_type swType, bangdb_count_type countType);
```

User can create entities to be counted by using this API. The swType refers to the type of sliding window. There are three options over here;

```
enum bangdb_window_type
{
        NON_SLIDING_WINDOW,
        SLIDING_WINDOW_SPAN,
        SLIDING_WINDOW_UNIT,
};
```

The first one is for non sliding window, which means the sliding time is infinite. Useful for doing counting for the infinite time period, which mean count all until stopped

The second one for spanned sliding window. Which means that counting is per span or per sliding window and there no further granularity. Which implies that the all counting is true for the whole span of the sliding window

The third one is for slotting the sliding window in to smaller units hence lot more granularity

3. add

```
int add(char *entityName, char *s, int len)
```

This allows user to pass in the string (for an entity) for which they want to count. Note that the count is per entity and user may define unique and non unique count for the entity. Using this API user will be able to count such entity

```
void add_create(char *entityName, char *s, int len, bangdb_window_type swType,
bangdb_count_type countType);
```

This API can be called if user wishes to add counting for entities which may not have already been added, hence if not present the first create the entity and then add count for it

4. remove entity

```
void removeEntity(char *name)
```

Use this to remove the entity from the sliding window for counting

5. list

```
char* list_count()
```

To get the list of all the items added for count for all entities. This returns json string with appropriate data

```
char *list_count2()
```

This returns simple string of items which has count greater than zero

6. count

```
int count(char *entityName)
```

```
int count(char *entityName, int span)
```

These APIs return the overall count for the given entity and within span

7. shutdown

```
void shutdown()
```

This shuts down the sliding window

TopK

This is another abstraction provided for addressing useful part of typical analysis. There are many scenarios where we would like to count unique or non unique top k items where k is configurable, for ex; top 30 pages, number of visits wise, top 10 customers with shopping carts by number of items in it etc..

User just needs to create the object of the type and use the API to deal with it, everything else is taken care by the abstraction

1. create

```
topk(char *topkName, int swSizeSec, int k, bool desc, char *uniqueBy)
```

This creates the topk object for the topkName. The swSiezeSec is parameter using which we can make it slide as well for the window period as defined by the parameter. The k defines the number of items to track (the 'k' in topk). UniqueBy parameter makes the counting unique by the parameter and finally desc is for sort direction

2. put

```
void put(long score, char *data, int datalen, char *uniqueParam)
```

The score is actual value which is used to decide the topk worthiness for any data for the given unique param.

3. GetTopK

```
resultset *getTopK(int k = 0)
```

```
char *getTopKJson(int k = 0)
```

The first API returns the cursor/resultset and next one returns json string

4. close

```
void close()
```

Few useful APIs

timestamp

useful timestamp related APIs

```
ULONG_T getCurrentTimeMicroSec()
ULONG_T getCurrentTimeMilliSec()
ULONG_T uniqueTimeStampMicroSec()
ULONG_T uniqueTimeStampMilliSec()
ULONG_T subTimeMicroSec(ULONG_T curTimeMicroSec, long nsec)
ULONG_T subTimeMilliSec(ULONG_T curTimeMilliSec, long nsec)
ULONG_T addTimeMicroSec(ULONG_T curTimeMicroSec, long nsec)
ULONG_T addTimeMilliSec(ULONG_T curTimeMilliSec, long nsec);
```

composite

APIs to create composite keys

```
char *makeComposite_long_long(ULONG_T k1, ULONG_T k2);
char *makeComposite_str_str(char *k1, char *k2);
char *makeComposite_str_str(void *k1, int l1, void *k2, int l2);
char *makeComposite_long_str(ULONG_T k1, char *k2);
char *makeComposite_long_str(ULONG_T k1, void *k2, int l2);
char *makeComposite_str_long(char *k1, ULONG_T k2);
char *makeComposite_str_long(void *k1, int l1, ULONG_T k2);
char *makeComposite(char *k, ...);
```

Logging

The BangDB has logger as well which can be used for general app logging. The BangDB also uses this for internal error logging. User can set the logging mechanism by setting the flag in bangdb.config

BANGDB_SIGNAL_HANDLER_STATE

Value 0 means logging to standard output and 1 to syslog (for linux, else for windows this would also log to standard output) and 2 for BangDB custom log. The value 2 is the recommended value for the logging as BangDB implements high performance logging mechanism for general purpose logging as well

The logging API is;

void bangdb_logger(const char *fmt, …)