# BangDB – API (Embedded Version 1.5)

CSharp (C#)

www.iqlect.com

The document covers the API for Bangdb, embeddable version. The API for C# are covered here.

# API for C#

## Database

1. To create database

```
Database(string dbName, string configpath, TransactionType transaction_type=TransactionType.DBTransactionNone, string dbpath = null)
```

The TransactionType by default is *DBTransactionNone* 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 database object, else program quits with error message

db_transaction_type has following values

*DBTransactionNone,*
*DBOptimisticTransaction,*
*DBPessimisticTransaction,*

As of now the *DBPessimisticTransaction* type is not supported

2. To get a table

```
Table GetTable(string tableName, DBOpenType flag = DBOpenType.Opencreate, TableEnv tblEnv = null)
```

```
WideTable GetWideTable(string tableName, DBOpenType flag = DBOpenType.Opencreate, TableEnv tblEnv = null)
```

```
Table GetPrimitiveTable(string tableName, BangDBPrimitiveDataType dataType = BangDBPrimitiveDataType.PrimitiveLong, DBOpenType flag = DBOpenType.Opencreate, TableEnv tblEnv = 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 DBOpenType 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

PersistType

*InmemOnly,*
*InmemPersist,*
*PersistOnly,*

IndexType

*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 (BangDBKeyType)

*NormalKey,*
*CompositeKey,*
*NormalKeyLong*

allow duplicates (boolean)

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

log (wal) state (WALState)

*Disabled,*
*Enabled,*
*DefaultConfig,*

BangDBLogType (private or shared log)

*SharedLog,*
*PrivateLog,*

log size in MB
size of log file in MB

auto commit (AutoCommitState)

*Disabled,*
Enabled,
DefaultConfig,

table type (BangDBTableType, default is NormalTable)

*NormalTable,*
*WideTable,*
*IndexTable, //not to be used by developers, only for BangDB*
*PrimitiveTableInt,*
*PrimitiveTableLong,*
*PrimitiveTableString*

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

table size hint ( TableSizeHint, 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 (BangDBKeySortMethod)

*Lexicograph,*
*QuasiLexicograph,*

sort direction (BangDBKeySortDirection)

*SortAscending,*
*SortDescending,*

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

It is good practice to use TableEnv 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 table object else null with error message

3. Close database

```
void CloseDatabase(DBCloseType flag = DBCloseType.Default)
```

The possible values for DBCloseType 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

4. Close table

```
int CloseTable(string tableName, DBCloseType flag = DBCloseType.Default)

int CloseTable(Table tbl, DBCloseType flag = DBCloseType.Default)

int CloseTable(WideTable wtbl, DBCloseType flag = DBCloseType.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

```
Transaction BeginTransaction()
```

Returns a Transaction object which should be used in all operations participating in the transaction

6. Committing a transaction

```
long CommitTransaction(ref Transaction txn)
```

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

7. Aborting a transaction

```
void AbortTransaction(ref Transaction txn)
```

## Table

1. Getting a connection

```
Connection GetConnection()
```

Returns a connection object else null with error message

2. Closing this table

```
int CloseTable(DBCloseType flag = DBCloseType.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 connection object else null with error message

2. Closing this table

```
int CloseTable(DBCloseType flag = DBCloseType.Default)
```

Returns 0 for successful close else -1 with error message

### 3. Adding/Creating Index

```
int AddIndex_Str(string idxName, int idxSizeByte, bool allowDuplicates)

int AddIndex_Num(string idxName, bool allowDuplicates)

int AddIndex(string idxName, TableEnv tenv)

int DropIndex(string idxName)
```

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 CloseAllConnection()
```

## **Connection** (for Normal Table)

### 1. Putting key value without transaction

```
long Put(long key, string val, InsertOptions flag)

long Put(long key, byte[] val, InsertOptions flag)

long Put(string key, ref DataVar dv, InsertOptions flag)

long Put(byte[] key, ref DataVar dv, InsertOptions flag)

long Put(byte[] key, byte[] val, InsertOptions flag)

long Put(string key, string val, InsertOptions flag)

long Put(string key, byte[] val, InsertOptions flag)

long Put(byte[] key, string val, InsertOptions flag)
```

DataVar 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 DataVar is type has following APIs;

Create DataVar

DataVar(int sizeInBytes)

The sizeInBytes tells how much buffer to pre allocate

Copying of buffer as following APIs

void Copy(string _buf, int len)

void Copy(byte[] _buf, int len)

Reading of the buffer

string Read()

string Read(int len)

Compare or equal

bool Equals(string _buf, int len)

bool Equals(byte[] _buf, int len)

bool Equals(byte[] buffer1, int offset1, byte[] buffer2, int offset2, int count)

There are others getters and setters exposed by the DataVar type

The InsertOptions has following options

*InsertUnique*,
*UpdateExisting*,
*InsertUpdate*,
*DeleteExisting*,

Return 0 for success else -1 with error message

2. Putting key value with transaction

```
long Put(string key, ref DataVar dv, InsertOptions flag, Transaction txn)
long Put(byte[] key, ref DataVar dv, InsertOptions flag, Transaction txn)
long Put(byte[] key, byte[] val, InsertOptions flag, Transaction txn)
long Put(string key, string val, InsertOptions flag, Transaction txn)
long Put(string key, byte[] val, InsertOptions flag, Transaction txn)
long Put(byte[] key, string val, InsertOptions flag, Transaction txn)
```

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

```
bool Get(long key, byte[] val)
bool Get(long key, out string vout)
bool Get(string key, out byte[] val)
bool Get(byte[] key, out string val)
```

bool Get(byte[] key, out byte[] val)

bool Get(string key, out string val)

bool Get(string key, ref DataVar dv)

bool Get(byte[] key, ref DataVar dv)

returns true or false as an indication of error and user should check for this

4. Getting value for a key with transaction

```
bool Get(string key, out byte[] val, Transaction txn)
```

bool Get(byte[] key, out string val, Transaction txn)

```
bool Get(byte[] key, out byte[] val, Transaction txn)
```

bool Get(string key, out string val, Transaction txn)

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

returns true or false as an indication of error and user should check for this

5. Deleting a key value without transaction

long Del(byte[] key)

```
long Del(string key)
```

long Del(long key)

Return 0 for success else -1 with error message

6. Deleting a key value without transaction

long Del(byte[] key, Transaction txn)

long Del(string key, Transaction txn)

Return >=0 for success else -1 with error message

7. Range query or scan without transaction

ResultSet Scan(long skey, long ekey, ScanFilter sf = null)

ResultSet Scan(byte[] skey, byte[] ekey, ScanFilter sf = null)

```
ResultSet Scan(string skey, string ekey, ScanFilter 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.

ScanFilter is provided to further add operators while doing the query and provide some limits on the returned resultset. The ScanFilter type is as follows

```
public class ScanFilter
{
    public ScanOperator skeyOp;
    public ScanOperator ekeyOp;
    public ScanLimitBy limitBy;
    public int limit;
    public ScanFilter()
    {
        skeyOp = ScanOperator.GTE;
        ekeyOp = ScanOperator.LTE;
        limitBy = ScanLimitBy.LimitResultSizeByte;
```

```
        limit = 2 * 1024 * 1024;
    }
}
```

The ScanOperator has following values

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

And ScanLimitBy has following values

```
LimitResultSizeByte,    //defines the bytes of data that should be returned (max)
LimitResultRow,  //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(byte[] skey, byte[] ekey, Transaction txn, ScanFilter sf = null)
```

```
ResultSet Scan(string skey, string ekey, Transaction txn, ScanFilter 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

```
long Count(byte[] skey, byte[] ekey, ScanFilter sf = null)
```

```
long Count(string skey, string ekey, ScanFilter sf = null)
```

```
long Count()
```

```
long Count(long sk, long ek, ScanFilter sf = null)
```

```
long Count(int sk, int ek, ScanFilter sf = null)
```

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

10. Setting auto commit

```
void SetAutoCommit(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

```
bool b = conn.Get(k, out v);                    //Not OK
```

we will have to either use the transactional API,

```
Transaction txn = db.BeginTransaction();
bool b = conn.Get(k, out v, txn);
```

Or, set the autocommit ON,

```
conn.SetAutocommit(true);
bool b = conn.Get(k, out v);                    //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

```
public bool Get(long key, ref long val);

public bool Get(int key, ref int val);

public bool Get(string key, ref long val);

public bool Get(byte[] key, ref long val);
```

2. put

```
public long Put(long key, long val, InsertOptions flag);

public long Put(int key, int val, InsertOptions flag);

public long Put(string key, long val, InsertOptions flag);

public long Put(byte[] key, long val, InsertOptions 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

```
public ResultSet ScanDoc(string idxName, string skey, string ekey, ScanFilter sf = null);

public ResultSet ScanDoc(string idxName, long skey, long ekey, ScanFilter sf = null);

public ResultSet Scan(string idxName, string skey, string ekey, ScanFilter sf = null);

public ResultSet Scan(string idxName, byte[] skey, byte[] ekey, ScanFilter sf = null);
```

2. put

```
public long PutDoc(string jsonStr);

public long PutDoc(string key, string jsonStr, InsertOptions flag);

public long PutDoc(long key, string jsonStr, InsertOptions flag)

public long Put(byte[] key, string val, string idxName, string idxVal);

public long Put(string key, string val, string idxName, string idxVal);
```

## 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 BeginReverse()
```

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 BeginReverse() before iterating and then use the same APIs for moving next or getting key and value

4. Getting the next key

```
bool GetNextKey(out byte[] key)

bool GetNextKey(out string key)

long GetNextKeyLong()

string GetNextKeyStr()
```

5. Getting the next val

```
bool GetNextVal(out byte[] val)

bool GetNextVal(out string val)

long GetNextValLong()

string GetNextValStr()
```

6. Last Evaluated Key

```
bool LastEvaluatedKey(out byte[] key)

bool LastEvaluatedKey(out string key)
```

long LastEvaluatedKeyLong()

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

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

bool SearchValue(byte[] key, out byte[] val)

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 AddDoc(ResultSet rs, string 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, string tableName, TableEnv tenv, int ttlSec)
```

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(string idxName, TableEnv tenv)
```

4. GetConnection

```
IntPtr GetConnection()
```

5. Put

```
int Put(string str, InsertOptions flag)
```

```
int Put(string str, string idxName, string idxKey)
```

6. Scan

```
ResultSet Scan(int period)
```

```
ResultSet Scan(int period, int lag)
```

```
ResultSet ScanFull()
```

```
ResultSet ScanRemaining(long fromTime, 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 above 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(string name, BangDBWindowType swType, BangDBCountType 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 BangDBWindowType
{
        NonSlidingWindow,
        SlidingWindowSpan,
        SlidingWindowUnit,
}
```

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(string entityName, string s)
```

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 AddCreate(string entityName, string s, BangDBWindowType swType, BangDBCountType 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(string name)
```

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

5. List

```
string ListCount_Json()
```

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

```
string ListCount_Str()
```

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(string name, int swSizeSec, int k, bool desc, string 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, string data, string 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)
```

```
string 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 GetCurrentTimeMicroSec()
```

```
ulong GetCurrentTimeMilliSec()
```

```
ulong UniqueTimeStampMicroSec()
```

```
ulong UniqueTimeStampMilliSec()
```

```
ulong GetCurrentTime()
```

```
ulong GetUniqueTimeStamp()
```

## Composite

APIs to create composite keys

```
string MakeComposite(string[] keys)

string MakeComposite(string k1, string k2)

string MakeComposite(long k1, long k2)

string MakeComposite(byte[] k1, byte[] k2)

string MakeComposite(string k1, long k2)

string MakeComposite(long k1, string k2)
```

## 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