BangDB **– API (Embedded Version 1.5)**

Java

www.iqlect.com

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

API for Java

Database

1. To create database

DatabaseImpl(String databasename, String configPath, TransactionType transaction_type, String dbPath)

The TransactionType by default is *TransactionNone* 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

TransactionType has following values

> *DB_MULTIOPS_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(String name, DBAccess openflag, TableEnv tenv)

WideTable getWideTable(String name, DBAccess openflag, TableEnv tenv)

Table getPrimitiveTable(String name, BangDBPrimitiveDataType dataType, DBAccess openflag, TableEnv tenv)

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 DBAccess 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 TableEnv 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 TableEnv then db config (defined in bangdb.config) values will apply to the table as well, but user can override the values by creating TableEnv 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)

> NORMAL_KEY,
> *COMPOSITE_KEY,*
> *NORMAL_KEY_LONG,*
> *KEY_TYPE_INVALID*

allow duplicates (boolean)

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

log (wal) state

> *false,*
> *true,*

LogType (private or shared log)

> *SHARED_LOG,*
> *PRIVATE_LOG,*
> *LogType_INVALID*

log size in MB

> size of log file in MB

auto commit (AutoCommitState)

> *false,*
> true,

table type (BangDBTableType, default is NORMAL_TABLE)

> *NORMAL_TABLE,*
> *WISE_TABLE,*
> *INDEX_TABLE, //not to be used by developers, only for BangDB*
> *PRIMITIVE_TABLE_INT,*
> *PRIMITIVE_TABLE_LONG,*
> *PRIMITIVE_TABLE_STRING,*
> *TABLE_TYPE_INVALID*

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)

bangdb, www.iqlect.com

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 (BangDBSortMethod)

> *LEXICOGRAPH,*
> *QUASI_LEXICOGRAPH,*
> *SORT_METHOD_INVALID*

sort direction (BangDBSortDirection)

> *SORT_ASCENDING,*
> *SORT_DESCENDING,*
> *SORT_DIRECTION_INVALID*

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(DBClose dbclose)`

The possible values for DBClose 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 before closing the database

4.   Close table

`int closeTable(Table tbl, DBClose dbclose)`

`int closeTable(WideTable tbl, DBClose dbclose)`

`int closeTable(String tableName, DBClose tbclose)`

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(Transaction txn);`

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

7. Aborting a transaction

`void abortTransaction(Transaction txn);`

Table

1. Getting a connection

`Connection getConnection()`

Returns a connection object else null with error message

2. Closing this table

`int closeTable(DBClose tableClose)`

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(DBClose tableClose)`

Returns 0 for successful close else -1 with error message

3.   Adding/Creating Index

```
int addIndex_str(String idx_name, int idx_size, boolean allowDuplicates)
```

```
int addIndex_num(String idx_name, boolean allowDuplicates)
```

```
int addIndex(String idx_name, 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 TableEnv

4.   Dumping the data

void 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

5.   Closing all active connections

```
int closeAllConnections()
```

Connection (for Normal Table)

1.   Putting key value without transaction

public long put(long key, String val, InsertOptions flag);

public long put(long key, byte[] val, InsertOptions flag);

public long put(String key, DataVar val, InsertOptions flag);

public long put(byte[] key, DataVar val, InsertOptions flag);

public long put(byte[] key, byte[] val, InsertOptions flag);

public long put(String 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 setBuffer(byte[] _buf)

Reading of the buffer

byte[] getBuffer()

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

public long put(String key, DataVar val, InsertOptions flag, Transaction txn);

public long put(byte[] key, DataVar val, InsertOptions flag, Transaction txn);

public long put(String key, String val, InsertOptions flag, Transaction txn);

```
public long put(byte[] key, byte[] 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

public String getStr(long key);

public byte[] getByte(long key);

public int get(String key, DataVar dv);

public int get(byte[] key, DataVar dv);

public String get(String key);

public byte[] get(byte[] key);

4. Getting value for a key with transaction

public String get(String key, Transaction txn);

public byte[] get(byte[] key, Transaction txn);

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

5. Deleting a key value without transaction

public long del(long key);

public long del(String key);

public long del(byte[] key);

Return 0 for success else -1 with error message

6. Deleting a key value without transaction

public long del(String key, Transaction txn);

public long del(byte[] key, Transaction txn);

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

7. Range query or scan without transaction

public ResultSet scan(long skey, long ekey, ScanFilter sf);

public ResultSet scan(String skey, String ekey, ScanFilter sf);

public ResultSet scan(byte[] skey, byte[] ekey, ScanFilter sf);

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()
        {
                reset();
        }

        public void reset()
        {
                skeyOp = ScanOperator.GTE;
                ekeyOp = ScanOperator.LTE;
                limitBy = ScanLimitBy.LIMIT_RESULT_SIZE;
                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

        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

public ResultSet scan(String skey, String ekey, Transaction txn, ScanFilter sf);

public ResultSet scan(byte[] skey, byte[] ekey, Transaction txn, ScanFilter sf);

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

```
public long count(long skey, long ekey, ScanFilter sf);
public long count(String skey, String ekey, ScanFilter sf);
public long count(byte[] skey, byte[] ekey, ScanFilter sf);
public long count();
```

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(boolean 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

String v = conn.get(k)   //Not OK

we will have to either use the transactional API,
        Transaction txn = db.beginTransaction()
        String v = conn.get(k, txn)

Or, set the autocommit ON,
        conn.setAutocommit(true)
        String 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

public Integer get(int key);

public Long get(long key);

public Long get(String key);

public Long get(byte[] key);

2. put

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

public long put(long key, long 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);

public ResultSet scanDoc(String idxName, long skey, long ekey, ScanFilter sf);

public ResultSet scan(String idxName, String skey, String ekey, ScanFilter sf);

public ResultSet scan(String idxName, byte[] skey, byte[] ekey, ScanFilter sf);

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(String key, String val, String idxName, String idxVal);

public long put(String key, byte[] val, String idxName, String idxVal);

public long put(byte[] key, String val, String idxName, String idxVal);

public long put(long key, String val, String idxName, String idxVal);

public long put(long key, byte[] val, String idxName, String idxVal);

public long put(byte[] key, byte[] val, String index_name, byte[] index_val);

ResultSet

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

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

4. Getting the next key

byte[] getNextKey()

long getNextKeyLong()

String getNextKeyStr()

First one returns byte array for key or null, second one is used for int/long data type and third one is for string

5. Getting the next val

byte[] getNextVal()

long getNextValLong()

String getNextValStr()

6. Last Evaluated Key

byte[] lastEvaluatedKey()

long lastEvaluatedKeyLong()

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

returns the byte array for key

7. Check if more data is to come

boolean 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

`byte[] searchValue(byte[] key)`

`byte[] searchValue(String key)`

returns the value for the given key

10. Freeing the resultset resource

`void clear()`

User must call this to clear the resource

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(ResultSetImpl rs, String orderBy)
```

```
void add(ResultSetImpl rs)
```

```
void append(ResultSetImpl rs)
```

```
void intersect(ResultSetImpl 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

```
SWTableImpl(DatabaseImpl 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

```
void 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

```
WideConnectionImpl getConnection()
```

5. Put

```
int put(String str, InsertOptions iop)
```

```
int put(String str, String idx, 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

```
SWEntityCountImpl(String entityName, int swTime, int swExpiry, int nEntity)
```

User will be able to create the sliding window for counting entities by calling the above constructor. The `entityName` 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
{
        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(String entityName, String str)
```

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 str, 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 listCountJson()
```

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

```
String listCountStr()
```

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

6. Count

```
int count(String entityName)
```

```
int count(String 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

   ```
   TopKImpl(String topkName, int swSizeSec, int k, boolean 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, int datlen, 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

   ```
   String getTopKJson(int k)
   ```

   ```
   ResultSet getTopK(int k)
   ```

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

4. Close

   ```
   void close()
   ```

Few useful APIs

Timestamp

```
Not supported separately, use the java API for now. Will be added in upcoming minor release
```

Composite

```
Not supported separately, concat two String with ":" to get the composite key. For example composite
key for k1 and k2 would be k1:k2. The API will be added in upcoming minor release
```

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