DBreeze Database Documentation.

DBreeze Database is a professional, open-source, multi-paradigm (embedded Key-Value store, objects, NoSq, Vector etc), multi-threaded, transactional and ACID-compliant database management system for .NET 3.5> / Xamarin MONO Android iOS / .NET Core 1.0> / .NET Standard 1.6> / Universal Windows Platform / .NET Portable ...for servers and internet-of-things... Made with C#

Document evolution.

This document evolves downside. All new features, if you have read the base document before, will be reflected underneath. New evolution always starts from a mark in format [year month day] - [20120521] - for the easy search.

Evolution history [yyyyMMdd]

20231103 DBreeze as an Embedding Vector Database / Similarity Search Engine / Clustering, based on KMeans++ Centroids Propagated Graph. Benchmarking. Mobile devices optimization.
20211227 Integrated CloneByExpressionTree and CPU time economy on huge amounts of object instantiations.
20211213 Reading of the same structured keys from different tables simultaneously. Multi_SelectForwardFromTo && Multi_SelectBackwardFromTo
20211208 Overloads for SelectBackwardFromTo and SelectForwardFromTo
20211005   Await inside DBreezeEngine transaction
20211004   Generic versions of DBreeze.Utils.MultiKeyDictionary and MultiKeySortedDictionary and some more addons
20210104 - Embedding Biser as a custom serializer for DBreeze
20180220 - Integrated binary/JSON serializer Biser.NET into v1.90 . Available via using DBreeze.Utils; Biser.

20120905 - Support of incremental backup.
20120628 - Row has property LinkToValue
20120601 - Storing inside of a row a column of dynamic data length. InsertDataBlock
           - Hash Functions of common usage. Fast access to long strings and byte arrays.
20120529 - Nested tables memory management. Nested Table Close(), controlling memory consumption.
           - Secondary Index. Direct key select.
20120526 - InsertDictionary/SelectDictionary InsertHashSet/SelectHashSet continuation.
20120525 - Row.GetTable().
           InsertDictionary/SelectDictionary InsertHashSet/SelectHashSet
20120521 - Fractal Tables structure description and usage techniques.
20120509 - Basic techniques description

[20120509]
Getting started.

DBreeze.dll contains fully managed code without references to other libraries. Current DLL size is around 470 KB. Start using it by adding its reference to your project. Don't forget DBreeze.XML from the Release folder to get VS IntelliSense help.

DBreeze is a disk based database system, though it also can work like in-memory storage.

Dbreeze doesn't have a virtual file system underneath and resides all working files in your OS file system, that's why you must instantiate its engine by supplying a folder name where all files can be located.

Main DBreeze **namespace** is **DBreeze**.

```
using DBreeze;

DBreezeEngine engine = null;

if(engine == null)
            engine = new DBreezeEngine(@"D:\temp\DBR1");
```

It's **important** in the **Dispose** function of your application or DLL to call DBreeze engine Dispose, to have graceful application termination.

```
if(engine != null)
     engine.Dispose();
```

Though, **DBreeze is resistant to loss of power**.

It's recommended to instantiate DBreeze engine as a static variable in the beginning of the application and dispose it in the end of the application work cycle (or just to exit without

dispose, together with the application)

After you have instantiated the engine there will be two options available for you, either to work with the database scheme or to work with the transactions.

Scheme.

You don't need to create tables via scheme, it's needed to make manipulations with already existing objects.

Deleting table:
engine.Scheme.DeleteTable(string userTableName)

Getting specific tables names:
engine.Scheme.GetUserTableNamesStartingWith(string mask)

Renaming table:
engine.Scheme.RenameTable(string oldTableName, string newTableName)

Checking if table exists:
engine.Scheme.IfUserTableExists(string tableName)

Getting physical path to the file holding the table:
engine.Scheme.GetTablePathFromTableName(string tableName)

Later more functions will be added there and their description here.

Transactions

In DBreeze all operations with the data, which resides inside of the tables, must occur inside of the transaction.

We open transaction like this:

```
using (var tran = engine.GetTransaction())
{

}
```

Please note, that it's **important** to dispose of a transaction after all necessary operations are done (using-statement makes it automatic).
Please note, that one transaction can be run only in **one** .NET managed thread and can not

be delegated to other threads.
Please note, that nested transactions are **not** allowed (parent transaction will be terminated)

During in-transactional operations different things can happen. That's why we **highly recommend** using try-catch blocks together with the transaction and **log exceptions** for future analysis.

```
try
{
    using (var tran = engine.GetTransaction())
    {
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

Table data types

Every table in DBreeze is a key/value storage. On the low level, keys and values represent arrays of bytes - byte[].

On the top level you can choose your own data type, from the allowed list, to be stored as a key or value.

There are some not standard data types in DBreeze, added for usability, they are accessible inside of DBreeze.DataTypes namespace.

```
using DBreeze.DataTypes;
```

Table data types. Key data types

Keys **can not** contain **NULLABLE** data types.

Note, that the key in the table is always unique.

Here is a list of available data types for the key:

Key data types

byte[]
int
uint
long
ulong
short
ushort
byte
sbyte
DateTime
double
float
decimal
string - *this one will be converted into byte[] using UTF8 encoding*
DbUTF8 - *this one will be converted into byte[] using UTF8 encoding*
DbAscii - *this one will be converted into byte[] using Ascii encoding*
DbUnicode - *this one will be converted into byte[] using Unicode encoding*
char
*G*uid

Value data types

byte[]
int
int?
uint
uint?
long
long?
ulong
ulong?
short
short?
ushort
ushort?
byte
byte?
sbyte
sbyte?
DateTime
DateTime?

double
double?
float
float?
decimal
decimal?
string - *this one will be converted into byte[] using UTF8 encoding*
DbUTF8 - *this one will be converted into byte[] using UTF8 encoding*
DbAscii - *this one will be converted into byte[] using Ascii encoding*
DbUnicode - *this one will be converted into byte[] using Unicode encoding*
bool
bool?
char
char?
Guid

And some more exotic data types like:

DbXML<T>
DbMJSON<T>
DbCustomSerializer<T>

they are used for storing objects inside of the value, we will talk about them later.

Table operations. Inserting data

All operations with the data, except operations which can be done via scheme, must be done inside of the transaction scope. By pressing tran. intellisense will give you a list of all possible operations. We start from inserting data into the table.

```csharp
public void Example_InsertingData()
{
        using (var tran = engine.GetTransaction())
        {
            tran.Insert<int, int>("t1", 1, 1);
            tran.Commit();
        }
}
```

In this example we have inserted data into the table with the name "t1". **Table** will be **created automatically**, if it doesn't exist.

Key type for our table is int 1, the value type of the table is also int (also 1).

After one or series of modifications inside of the transaction we must either Commit them or Rollback them.

Note, **Rollback** function will **automatically run** in the **transaction Dispose** function, so all not committed modifications of the database inside of transaction will be automatically rolled-back.

You can be sure that this modification will not be applied to the table, but nevertheless an empty table will be created, if it doesn't exist before.

```
using (var tran = engine.GetTransaction())
        {
tran.Insert<int, int>("t1", 1, 1);
//NO COMMIT
}
```

We don't store in the table data types, which you assume must be there, the table holds only byte arrays of keys and values and only on the upper level acquired byte[] will be converted into keys or values of the appropriate data types from generic constructions.

You can modify more than one table inside of the transaction.

```
using (var tran = engine.GetTransaction())
{
    tran.Insert<int, int>("t1", 1, 1);
    tran.Insert<uint, string>("t2", 1, "hello");

tran.Commit();
//or
//tran.Rollback();

tran.Insert<int, int>("t1", 2, 1);
                    tran.Insert<uint, string>("t2", 2, "world");

tran.Commit();

            }
```

Commits and Rollbacks

Used Commit or Rollback will be applied to all modifications inside of the transaction. If something happens during Commit all data will be automatically rolled-back for all modifications.

The only acceptable reason for Rollback failure can be the damage of the physical storage, and exceptions in the rollback procedure will bring the database to the not operable state.

DBreeze database, after its start, checks transactions journal and restores tables into their previous state, so there should be no problems with the power loss or any other accidental software termination in any process execution point.

DBreeze database is fully [ACID](#) compliant.

Commit operation is always very **fast** and takes the same amount of time independent of the quantity of modifications made.

Rollback can take **longer**, depending upon the **quantity of data and character of modifications,** which were made within the database.

Table operations. Updates

Update key operation is the same as insert operation

```
tran.Insert<int, int>("t1", 2, 1);
tran.Insert<int, int>("t1", 2, 2);
```

We have updated key 2 and set up new value 2.

Table operations. Bulk operations

If you are going to insert or update a big data set then first execute insert, update, remove command as many times as you need and then call tran.Commit();

Calling tran.**Commit** after every operation, **will not make the table physical file bigger** but will take more time then one Commit after all operations.

```
using (var tran = engine.GetTransaction())
{
            //THIS IS FASTER
        for(int i=0;i<1000000;i++)
         {
            tran.Insert<int,int>("t1",i,i)
         }

      tran.Commit();

//THIS IS SLOWER

 for(int i=0;i<1000000;i++)
 {
       tran.Insert<int,int>("t1",i,i)
```

```
        tran.Commit();
 }
              }
```

Table operations. Random keys while bulk insert.

Dbreeze algorithms are built to work with maximum efficiency while inserting in bulk sorted ascending data.

```
 for(int i=0;i<1000000;i++)
 {
      tran.Insert<int,int>("t1",i,i);
 }

                tran.Commit();

                //or

                DateTime dt=DateTime.Now;

 for(int i=0;i<1000000;i++)
 {
      tran.Insert<DateTime, int >("t1",dt,i);
      dt=dt.AddSeconds(7);
 }

                tran.Commit();
```

The above code will be executed in 1.5 seconds (year 2015, HDD).

If you start to insert data in random order it can take a bit longer. That's why, if you have an in-memory big data set, before saving it to the database, sort it ascending in-memory by key and insert after that, it will speed up your program.

If you make a copy from other databases to DBreeze, take a chunk (e.g. 1 MLN records), sort it in memory by key ascending, insert into DBreeze, then take another chunk.. and so on.

Table operations. Partial Insert or Update

In DBreeze **maximum key length** in bytes is 65535 (UInt16.MaxValue) and maximum **value length** is 2147483647 (Int32.MaxValue).

It's not possible to save as a value byte array bigger than 2GB. For bigger data elements we will have to develop in the future other strategies (read DataBlocks later).

In DBreeze we have the ability of a partial value update or insert. It's possible because

values are stored as byte[]. It doesn't matter which data type is stored already in the table, you can always access it and change it as a byte array.

DBreeze has a special namespace inside, which allows you to easily work with byte arrays.

using DBreeze.Utils;

Now you can convert any standard data type into byte array and back.

We will achieve the same effect in all following records:

```
tran.Insert<int, int>("t1", 10, 1);
//or
tran.Insert<int, byte[]>("t1", 10, ((int)1).To_4_bytes_array_BigEndian());
//or
tran.Insert<int, byte[]>("t1", 10, new byte[] {0x80, 0x00, 0x00, 0x01});
```

Above instructions can be run one by one and will bring to the result then under key 10 we will have value 1.

And the same result we will achieve having run 4 following instructions:

```
tran.InsertPart<int, byte[]>("t1", 10, new byte[] { 0x80 }, 0);
tran.InsertPart<int, byte[]>("t1", 10, new byte[] { 0x00 }, 1);
tran.InsertPart<int, byte[]>("t1", 10, new byte[] { 0x00 }, 2);
tran.InsertPart<int, byte[]>("t1", 10, new byte[] { 0x01 }, 3);

//or the same
tran.InsertPart<int, byte[]>("t1", 10, new byte[] { 0x80 ,0x00}, 0);
tran.InsertPart<int, byte[]>("t1", 10, new byte[] { 0x00 ,0x01}, 2);
```

The fourth parameter of tran.InsertPart is exactly the index from which we want to insert our byte[] array.

This **technique** can be used if we think about the value as about the **set of columns** of the known length, like in **standard SQL databases**, and gives an ability to change every column separately, without changes in other parts of the values.

Note, you can always **switch to byte[] data type** in values and in keys

```
tran.Insert<int, int>
//or
tran.Insert<int, byte[]>
```

if it's interesting for you.

Note, If you want to insert or update the value starting from the **index which is bigger then current value** length, the empty space will be filled with byte[] { 0 }.

We didn't have before key 12 and now we are executing following commands:

```
tran.InsertPart<int, byte[]>("t1", 12, new byte[] { 0x80 }, 0);
tran.InsertPart<int, byte[]>("t1", 12, new byte[] { 0x80 }, 10);
```

Value as byte[] will look like this:

"0x80 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x80"

Note, Dbreeze will try to use the **same physical file space while record update,** if existing record length is suitable for this.

Table operations. Data Fetching. Select.

Method tran.Select is designed for getting one single key:

```
using (var tran = engine.GetTransaction())
        {
        tran.Insert<int, int>("t1", 10, 2);
            tran.Commit();

            var row = tran.Select<int, int>("t1", 10);
                //or will work also good
            var row = tran.Select<int, byte[]>("t1", 10);

        }
```

After select you must supply in generic format data types for the key and value.

In our case, we want to read from table "t1" key of type int (its value 10).

Select always returns a value of type DBreeze.DataTypes.Row.

We can **start to visualize** the key value **only after checking** if the table has such value inside.

Row has **property Exists**:

```
using (var tran = engine.GetTransaction())
```

```
        {
                tran.Insert<int, int>("t1", 10, 2);
                    tran.Commit();

                    var row = tran.Select<int, int>("t1", 10);

                    byte[] btRes = null;
                    int res=0;
                    int key=0;

                    if (row.Exists)
                    {
                            key = row.Key;
                            res = row.Value;

                        //btRes will be null, because we have only 4 bytes
                            btRes = row.GetValuePart(12);
                         //btRes will be null, because we have only 4 bytes
                            btRes = row.GetValuePart(12, 1);
                //will return 4 bytes
 btRes = row.GetValuePart(0);
                //will return 4 bytes
 btRes = row.GetValuePart(0,4);
                    }
        }
```

So, if the row exists, we can start to fetch its key (**row.Key**), full record **row.Value** (it will be automatically converted from byte[] to the data type, which you gave while forming Select). And independent from the record data type, Row has a method **GetValuePart** with overloads which will help you to get value partially and always as byte[]. DBreeze.Utils extensions can help to convert values to other data types.

If we had in the value, starting from index 4 stored some kind of ulong, which resides 8 bytes, we can say:

ulong x = row.GetValuePart(4,8).To_UInt64_BigEndian();

Note that DBreeze.Utils conversion algorithms are exactly sharpened for DBreeze data types, because they create sortable byte[] sequences in comparison with .NET built in byte[] conversion functions.

Table operations. Data Fetching. NULL

tran.Insert<int,int?>("t1",10,null);

var row = tran.Select<int,int?>("t1",10);

if(row.Exists)

```
{
        int? val = row.Value; //val will be null
}
```

Table operations. Data Fetching. Order by. Order by Descending.

When **Dbreeze stores data** in the table it's **automatically** stored in the **sorted** order. That's why all range selects are very fast. This example is taken from satellite project integrated into DBreeze solution, which is called VisualTester from class DocuExamples:

```csharp
public void Example_FetchingRange()
{
            engine.Scheme.DeleteTable("t1");

            using (var tran = engine.GetTransaction())
            {
            DBreeze.Diagnostic.SpeedStatistic.StartCounter("INSERT");

        DateTime dt = DateTime.Now;
                for (int i = 0; i < 1000000; i++)
                    {
                                tran.Insert<DateTime, byte?>("t1", dt,
null);
                        dt = dt.AddSeconds(7);
                    }

                tran.Commit();

            DBreeze.Diagnostic.SpeedStatistic.StopCounter("INSERT");
            DBreeze.Diagnostic.SpeedStatistic.PrintOut(true);
            DBreeze.Diagnostic.SpeedStatistic.StartCounter("FETCH");

            foreach (var row in tran.SelectForward<DateTime,
byte?>("t1"))
                {
                        //Console.WriteLine("K: {0}; V: {1}",
row.Key.ToString("dd.MM.yyyy HH:mm:ss"), (row.Value == null) ? "NULL" :
row.Value.ToString());
                }

            DBreeze.Diagnostic.SpeedStatistic.StopCounter("FETCH");

                DBreeze.Diagnostic.SpeedStatistic.PrintOut(true);
        }

}
```

A small benchmark for this procedure:
INSERT: 10361 ms; 28312951 ticks

All range selects methods in DBreeze return IEnumerable<Row<TKey,TValue>>, so they can be used in foreach statements.

If you want, you can break from foreach at any moment.

To limit the quantity of the data you can use, either break iteration or use **Take** statement**:**

foreach (var row in tran.SelectForward<DateTime, byte?>("t1").**Take**(100))

SelectForward - starts from the first key and iterates forward to the last key in sorted **ascending** order.

SelectBackward - starts from the last key and iterates backward to the first key in sorted **descending** order.

Transaction has more self-explained methods:

IEnumerable<Row<TKey, TValue>> **SelectForwardStartFrom**<TKey, TValue>(string tableName,TKey key, bool includeStartFromKey) - **Note, if key is not found then it starts from the next available key forward in ascending order, idea of non-existing supplied parameter concerns all iteration methods.**

SelectBackwardStartFrom<TKey, TValue>(string tableName, TKey key, bool includeStartFromKey) - iterates from the given key down in descending order.

SelectForwardFromTo<TKey, TValue>(string tableName, TKey startKey, bool includeStartKey, TKey stopKey, bool includeStopKey)

SelectBackwardFromTo<TKey, TValue>(string tableName, TKey startKey, bool includeStartKey, TKey stopKey, bool includeStopKey)

DON'T USE **LINQ** after **SelectForward or SelectBackward while filtering by key**, like this:

tran.SelectForward<int,int>("t1").Where(r=>r.Key > 10).Take(10)

Because it will work **much much much slower** than specially sharpened methods, used instead:

tran.**SelectForwardStartFrom**<int,int>("t1",10,false).Take(10)

And finally two more special methods:

SelectForwardStartsWith<TKey, TValue>(string tableName, TKey startWithKeyPart)

and

SelectBackwardStartsWith<TKey, TValue>(string tableName, TKey startWithKeyPart)

You remember that all data types will be converted into byte[].

So if in table we have keys

byte[] {0x12, 0x15, 0x17}
byte[] {0x12, 0x16, 0x17}
byte[] {0x12, 0x15, 0x19}
byte[] {0x12, 0x17, 0x18}

then
SelectForwardStartsWith<byte[],int>("t1",new byte[] {0x12})

will return us all keys
SelectForwardStartsWith<byte[],int>("t1",new byte[] {0x12, 0x15})

will return us only 2 keys

byte[] {0x12, 0x15, 0x17}
byte[] {0x12, 0x15, 0x19}

SelectBackwardStartsWith<byte[],int>("t1",new byte[] {0x12, 0x15})

will return us only 2 keys in descending order

byte[] {0x12, 0x15, 0x19}
byte[] {0x12, 0x15, 0x17}

SelectForwardStartsWith<byte[],int>("t1",new byte[] {0x12, 0x17})

will return us 1 key
byte[] {0x12, 0x17, 0x18}

and
SelectForwardStartsWith<byte[],int>("t1",new byte[] {0x10, 0x17})

will return nothing.

Having this idea we can effectively work with strings:

tran.Insert<string,string>("t1","w","w");
tran.Insert<string,string>("t1","ww","ww");
tran.Insert<string,string>("t1","www","www");

then

SelectForwardStartsWith<string,string>("t1","ww")

will return us

"ww"
"www"

and **SelectBackwardStartsWith**<string,string>("t1","ww")

will return us

"www"
"ww"

Table operations. Skip

In Dbreeze we have ability to start iterations after Skipping some other keys:

this command skips "skippingQuantity" elements and then starts enumeration in ascending order:

SelectForwardSkip<TKey, TValue>(string tableName, ulong skippingQuantity)

this command skips "skippingQuantity" elements backward and then starts enumeration in descending order:

IEnumerable<Row<TKey, TValue>> **SelectBackwardSkip**<TKey, TValue>(string tableName,ulong skippingQuantity)

this command skips "skippingQuantity" elements from the specified key (if key is not found

then next one after it will be taken as skipped 1) and then starts enumeration in ascending order:

SelectForwardSkipFrom<TKey, TValue>(string tableName, TKey key, ulong skippingQuantity)

this command skips "skippingQuantity" elements backward from the specified key and then starts enumeration in descending order:

SelectBackwardSkipFrom<TKey, TValue>(string tableName, TKey key, ulong skippingQuantity)

Note, that skip needs to iterate via keys, to calculate exact skipping quantity. That's why developers have always to take into consideration the idea of finding compromise between speed and skipping quantity. Skipping 1 MLN, of elements in **any direction** starting from **any key** will take 4 seconds with Intel i7 8 cores and SCSI drive 8GB RAM (year 2012). Skip of 100 000 records will take 400 ms, 10 000 will take 40 ms respectively.

So, if you are going to implement grid paging, then just remember first shown in the grid key and then skip from it the quantity shown in the grid elements using **SelectForwardSkipFrom** or **SelectBackwardSkipFrom**.

Table operations. Count.

For getting **Table** records **quantity** use:

ulong cnt = tran.**Count**("t1");

Count is calculated while inserting and removing operations and is always available.

Table operations. Max.

var row = tran.**Max**<int, int>("t1");

if (row.Exists)
{
        //etc...
}

Table operations. Min.

var row = tran.**Min**<int, int>("t1");

```
if (row.Exists)
{
        //etc...
}
```

Table operations. Reading from non-existing table

If you try to read from a non-existing table, this **table will not be created** in the file system.

tran.Count will return 0
tran.Select, tran.Min, tran.Max will return row with row.Exists == false

Range selects like tran.SelectForward etc. will return nothing in your foreach statement.

Table operations. Removing keys

To **remove one key** use

```
tran.RemoveKey<int>("t1",10)
tran.Commit();
```

To **Remove all keys** use
tran.RemoveAllKeys(string tableName, bool withFileRecreation)

Note, if the **withFileRecreation** parameter is set to **true**, then we **don't** need to **Commit** this modification, it will be done automatically. The file who holds the table will be re-created.

Note, if the **withFileRecreation** parameter is set to **false,** the old data will be not visible any more, but the old information will still reside in the table. We **need Commit** after this modification.

Table operations. Change key

We have an ability to change the key.

After these commands:

```
tran.Insert<int,int>("t1",10,10);
tran.ChangeKey<int>("t1", 10, 11);
tran.Commit();
```

we will have in the table one key 11 with the value 10.

After these commands:

tran.Insert<int,int>("t1",10,10);
tran.Insert<int,int>("t1",11,11);
tran.ChangeKey<int>("t1", 10, 11);
tran.Commit();

we will have in the table one key 11 with the value 10. (old value for the key 11 will be lost)

Storing objects in the database

For storing objects in the table we have 3 extra data types which are accessible via DBreeze.DataTypes namespace.

DbXML<T> - will automatically use built-in .NET XML serializer and deserializer for objects. Slower than others in both operations furthermore data resides much more physical space, then others.

DbMJSON<T> - Microsoft JSON, will automatically use built-in .NET JSON (System.Web.Script.Serialization.JavaScriptSerializer) serializer and deserializer for objects. Much better than XML but not as good as the serializer provided by https://github.com/rpgmaker/NetJSON.

DbCustomSerializer<T> - gives you the ability to attach your own serializer like https://github.com/rpgmaker/NetJSON .

To attach JSON.NET, download it, refer to your project and fill some lines:

DBreeze.Utils.CustomSerializator.Serializator = JsonConvert.SerializeObject;
DBreeze.Utils.CustomSerializator.Deserializator = JsonConvert.DeserializeObject;

Now you can use serialization and deserialization provided by JSON.NET.

But if you don't want to use JSON.NET, try Microsoft JSON. It's about 40% slower on deserialization and 5-10% slower on serialization then JSON.NET.

Use all of them in following manner:

```
public class Article
{
        public uint Id { get; set; }
        public string Name { get; set; }
}
```

```csharp
public void Example_InsertingObject()
{
        engine.Schema.DeleteTable("Articles");

        using (var tran = engine.GetTransaction())
        {
        tran.SynchronizeTables("Articles");

            uint identity = 0;

            var row = tran.Max<uint, byte[]>("Articles");

            if (row.Exists)
                identity = row.Key;

            identity++;

            Article art=new Article()
            {
                Id = identity,
                Name = "PC"
            };
            tran.Insert<uint, DbMJSON<Article>>("Articles", identity,
 art);

            tran.Commit();
        }
}
```

Note, DbMJSON, DbXML, DbMJSON,DbCustomSerializer have overloaded operator and you can specify art without saying new DbMJSON<Article>, just say art:

```csharp
tran.Insert<uint, DbMJSON<Article>>("Articles", identity, art);
//or
tran.Insert<uint, DbXML<Article>>("Articles", identity, art);
//or
tran.Insert<uint, DbCustomSerializer<Article>>("Articles", identity, art);
```

Getting objects:

```csharp
 foreach (var row in tran.SelectForward<uint,
DbMJSON<Article>>("Articles").Take(10))
            {
        //Note row.Value will return us DbMJSON<Article>
                        //row.Value
        //But we need Article
            //Article a = row.Value.Get
        //Or its serialized representation
```

```
                    //string aSerialized = row.Value.SerializedObject

                }
```

Multi-threading

In Dbreeze tables are always accessible for parallel READ of last committed data from multiple threads.

Note, while **one thread** is **writing data** into the table, **other threads** will **not** be **able** to **write** data in the same table (table lock), till the writing thread releases its transaction, they will wait in a queue.

Note, while **one thread** is **writing data** into the table, **other threads can** in parallel **read** already **committed data**.

Note, if **one** of the **threads** needs, inside of the transaction, to **read** data from the tables and it wants to be sure that till the end of transaction **other threads** will **not modify** the data, this thread must **reserve tables for synchronized read**.

```
 using (var tran = engine.GetTransaction())
{
                tran.SynchronizeTables("table1", "table2");

 }
```

Transaction also has a method for tables synchronization.

tran.SynchronizeTables

This method has overloads and you can supply as parameters: List<string> or params string[].

SynchronizeTables can be run only once inside of the transaction.

All **reads** can be divided on **two categories** by usage type:

- **Read for reporting**
- **Read for modification**

Based on this idea the whole multi-threaded layer is built.

Multi-threading. Read for reporting

If you think that there is no necessity to block table(s) and other threads could write data in parallel just don't use tran.SynchronizeTables.

This technique is applicable in all reporting cases. If a user needs to know his bank account state, we don't need to block the table with account information, just read the account state and return it. Doesn't matter that at this moment his account state is changing - it's a question of a moment. If a user requests his account state in 5 minutes he will get an already modified account.

There are some things which must be understood.

For example we make iteration via table Items, because someone has requested its full list.

Let's assume that there are 100 items

List<Item> items=new List<Item>();

foreach(var row in tran.SelectForward<ulong, DbMJSON<Item>>("Items"))
{
        items.Add(row.Value.Get);

        //we have iterated over 50 items and in this moment other thread deleted itemId 1 and committed transaction
        //Result: it's a question of the moment this item will be added to the final List, it doesn't matter in this case.

        //we have iterated already 75 items and in this moment other thread deleted itemId 90 and committed transaction
        //after 89 we will get item 91
        //Result: it's a question of the moment, item 90 will not be added to the final List, it doesn't matter in this case.

}

And if you want to be sure that other threads will not modify "Items" table, while you are fetching the data, use

tran.SynchronizeTables("Items");

If you take a row from a table, always check if it exists.

If your data projection is spread among many tables, first get all pieces of the data from different tables, always checking if row.Exists, in case of direct selects, and only when you have a full object constructed then return it to the final projection as a ready element.

Note if you have received a row and it exists. It doesn't mean that you have already acquired the value. Value will be read only when you choose property row.Value (lazy value loading). If another thread removes value in between, after you have acquired the row, but still didn't acquire value, - then value will be returned in any case, because after removing data still stays on the disk, only keys are marked as deleted. And this behavior for not synchronized reading should be ok, because it's a question of the moment.

If you have acquired a row and it exists in one thread, now you are going to get the value, but at this moment another thread updates the value, then your thread will receive updated value.

In case if your thread is going to retrieve value and in this moment DBreeze.Scheme deletes table - then inside of transaction exception will be raised, controlled by try-catch integrated into using statement.

The same will happen if another thread executes tran.RemoveAllKeys("your reading table", true - withFileRecreation). Your reading thread will get an exception inside of the transaction. But all will be ok if other threads remove data without file re-creation, if tran.RemoveAllKeys("your reading table", false- withFileRecreation).

You must use Scheme.DeleteTable, Scheme.RenameTable and tran.RemoveAllKeys with table re-creation semantically.

Either in the constructor, after engine initialization, or for temporary tables, which are used for sub-computation with the help of a database, and definitely only by one thread. For tables which are under read-write pressure, it is better to use tran.RemoveAll(false) and then one day to compact this table by copying existing values into the new table, and renaming the new table to the old table.

Tables copying / compaction

Copying of the data better to make it on byte[] level, it will be faster then to cast and serialize / deserialize objects.

If you had table Articles <ulong, DbMJSON<Article>>

Copy it like this:

```
foreach(var row in tran.SelectForward(<byte[],byte[]>("Articles")))
{
        tran.Insert<byte[],byte[]>("Articles Copy",row.Key, row.Value);
}

tran.Commit();
```

then you can rename old table Scheme.RenameTable("Articles Copy","Articles");

and go on to work with Article table

```
foreach(var row in tran.SelectForward(<long, DbMJSON<Article>>("Articles")))
{
        ...
}
```

Note, we create a foreach loop which reads from one table and after that writes into the other table. From HDD point of view we make such operation:
R-W-R-W-R-W-R-W …..

If you have a mechanical HDD, its head must always move between two files to complete this operation, which is not so efficient.

To increase performance of the copy procedure we need following sequence:

R-R-R-R-W-R-R-R-R-W-R-R-R-R-W ….

So, first we read to the memory a big chunk (1K/10K/100K/1MLN of records) and then sort it by key in ascending order and insert it in bulk to the copy table.

Dictionary<TKey,TValue> will not be able to sort byte[]. For this we need to construct hash-string using DBreeze.Utils:

byte[] bt=new byte[]{0x08, 0x09};

string hash = bt.ToBytesString();

then put this hash in a key for the Dictionary. Copy procedure with
R-R-R-R-W-R-R-R-R-W-R-R-R-R-W ….
sequence:

```
using DBreeze.Utils

int i = 0;
int chunkSize = 100000;
```

```csharp
Dictionary<string,KeyValuePair<byte[],byte[]>> cacheDict=new
Dictionary<string,KeyValuePair<byte[],byte[]>>();

foreach(var row in tran.SelectForward(<byte[],byte[]>("Articles")))
{
        cacheDict.Add(
row.Key.ToBytesString()
,new KeyValuePair<byte[],byte[]>
(
     row.Key,
     row.Value
)
     );


        i++;



     if(i == chunkSize)
     {
             //saving sorted values to the new table in bulk
             foreach (var kvp in cacheDict.OrderBy(r=>r.Key))
{
tran.Insert<byte[],byte[]>("Articles Copy",kvp.Value.Key, kvp.Value.Value);
             }

             cacheDict.Clear();
             i=0;
     }
}

//If something left in cache - flush it
foreach (var kvp in cacheDict.OrderBy(r=>r.Key))
{
tran.Insert<byte[],byte[]>("Articles Copy",kvp.Value.Key, kvp.Value.Value);
}
cacheDict.Clear();

tran.Commit();
```

Note, actually we don't need to sort dictionary, because SelectForward from table Articles gives us already sorted values and in sorted sequence they will migrate into cache-Dictionary, so our complete code will look like this:

```csharp
int i = 0;
int chunkSize = 100000;
Dictionary<byte[],byte[]> cacheDict=new Dictionary<byte[],byte[]>();
```

```
foreach(var row in tran.SelectForward(<byte[],byte[]>("Articles")))
{
        cacheDict.Add(row.Key,row.Value)
         i++;



      if(i == chunkSize)
      {
             //saving sorted values to the new table in bulk
             foreach (var kvp in cacheDict)
{
tran.Insert<byte[],byte[]>("Articles Copy",kvp.Key, kvp.Value);
             }

             cacheDict.Clear();
             i=0;
      }
}

//If something left in cache - flush it
foreach (var kvp in cacheDict)
{
tran.Insert<byte[],byte[]>("Articles Copy",kvp.Key, kvp.Value);
}
cacheDict.Clear();

tran.Commit();
```

Multi-threading. Read for modification

This technique is used when you need to get data (select) before modification (insert or update etc.):

```
private bool AddMoneyOnAccount(uint userId, decimal sum)
{
using (var tran = engine.GetTransaction())
{
        try
        {

string tableUserInfo = "UserInfo" + userId;

tran.SynchronizeTables(tableUserInfo);

//after SynchronizeTables, be sure that none of the other threads will write in
table tableUserInfo, till the transaction will be released.

//now we read the state of the user account
```

```csharp
var row = tran.Select<string,decimal>(tableUserInfo ,"Account");

decimal accountState = 0;

if(row.Exists)
      accountState = row.Value;

//now we change the sum of the user's account

accountState  += sum;

tran.Insert<string,decimal>(tableUserInfo, "Account", accountState);

tran.Commit();

            }
            catch (Exception ex)
            {
                    Console.WriteLine(ex.ToString());
                return false;
            }
  }

return true;
}
```

Table WRITE, Resolving Deadlock Situation

If we write only in one table inside of transaction and for other tables use unsynchronized read, we don't need to use **SynchronizeTables**

```csharp
using (var tran = engine.GetTransaction())
{
    tran.Insert<int,int>("t1",1,1);

}
```

But when we have inserted/updated/Removed a key in the table, DBreeze will automatically block the whole table for Write, like SynchronizeTables("t1") would be used, till the end of the transaction.

In following example, transaction first blocks table "t1" and then "t2"

```csharp
using (var tran = engine.GetTransaction())
```

```
    {
        tran.Insert<int,int>("t1",1,1);

        tran.Insert<int,int>("t2",1,1);

    }
```

Imagine, the we have parallel thread which writes in the same tables but in other sequence:

```
    using (var tran = engine.GetTransaction())
    {
        tran.Insert<int,int>("t2",1,1);

        tran.Insert<int,int>("t1",1,1);

    }
```

Thread 2 has blocked table "t2", which is going to be read by Thread 1, and Thread 1 has blocked table "t1", which is going to be read by Thread 2.

Such a situation is called deadlock.

Dbreeze automatically drops one of these threads with Deadlock Exception, and the other thread will be able successfully finish its job.

But this is only a part of the solution. To make the program deadlock safe use in both threads **SynchronizeTables** construction:

Thread 1:

```
    using (var tran = engine.GetTransaction())
    {
        tran. SynchronizeTables ("t1","t2");

        tran.Insert<int,int>("t1",1,1);

        tran.Insert<int,int>("t2",1,1);

    }
```


Thread 2:

```
    using (var tran = engine.GetTransaction())
    {
```

```
                tran. SynchronizeTables ("t1","t2");

                 tran.Insert<int,int>("t2",1,1);

                 tran.Insert<int,int>("t1",1,1);


            }
```

Both threads will be executed without exceptions, one by one - absolute protection from the deadlock situation.

Table WRITE, READ or SYNCHRO-READ, Data visibility scope

In the following example we read a row from table "t1".

```
        using (var tran = engine.GetTransaction())
        {
                var row = tran.Select<int,int>("t1",1);
        }
```

We didn't use tran.SynchronizeTables construction and we didn't write to this table before, so we will see only last committed data, even if another thread is changing the same data in parallel, this transaction will receive only last committed data for this table.

But everything changes when transaction has a table in modification list:

```
        using (var tran = engine.GetTransaction())
        {
                tran.Insert<int,int>("t1",1,157);
            //Table "t1" is in modification list of this transaction and all reads from
        this table automatically return actual data, even before commit

            //this row.Value will return 157
                var row = tran.Select<int,int>("t1",1);
        }
```

All reads of the table (only inside current transaction), if it's in modification list (by SynchronizeTables or just insert/update/remove) will return modified values even if the data was not committed yet:

```
 using (var tran = engine.GetTransaction())
 {
   tran.Insert<int,int>("t1",1,99);
   tran.Commit();
```

```
    }


    using (var tran = engine.GetTransaction())
    {

    //row.Value will return 99 like other parallel threads which read
    this table
        var row = tran.Select<int,int>("t1",1);
    //but this thread wants also to modify this table
     tran.Insert<int,int>("t1",1,117);

    //row.Value will return 117 (other threads will see 99)
        var row = tran.Select<int,int>("t1",1);

    tran.RemoveKey("t1",1);

    //row.Exists will be false (other threads will see 99)
        var row = tran.Select<int,int>("t1",1);

    /tran.Insert<int,int>("t1",1,111);

    //row.Value will return 111 (other threads will see 99)
    var row = tran.Select<int,int>("t1",1);

    tran.Commit();

    //row.Value will return 111 (other threads will see 111)
    var row = tran.Select<int,int>("t1",1);
    }
```

Table Synchronization by PATTERN

Because in the NoSql concept we have to deal with many tables inside of one transaction, DBreeze has special constructions for table locking. All these constructions are available via tran.SynchronizeTables.

Again, tran.SynchronizeTables can be used only once inside of any transaction before any modification command, but can be used after read commands:

ALLOWED:

```
    using (var tran = engine.GetTransaction())
    {
     tran.SynchronizeTable("t1");
```

```
    tran.Insert<int,int>("t1",1,99);
    tran.Commit();
}
using (var tran = engine.GetTransaction())
{
 tran.SynchronizeTable("t1","t2");
  tran.Insert<int,int>("t1",1,99);
  tran.Insert<int,int>("t2",1,99);
  tran.Commit();
}

using (var tran = engine.GetTransaction())
{
        List<string> ids=new List<string>();
        foreach(var row in tran.SelectForward<int,int>("Items"))
{
        ids.Add("Article" +row.Value.ToString());
}

  tran.SynchronizeTable(ids);

  tran.Insert<int,int>("t1",1,99);
  tran.Commit();
}
```

Note, it's possible to insert data into tables which were not synchronized by SynchronizeTable

```
using (var tran = engine.GetTransaction())
{
 tran.SynchronizeTable("t1");
  tran.Insert<int,int>("t1",1,99);
  tran.Insert<int,int>("t2",1,99);
  tran.Commit();
}
```

But this is better to use for temporary tables, for avoiding deadlocks. To add uniqueness to the table name (temporary table name) add ThreadId:

```
using (var tran = engine.GetTransaction())
{
        try{
 tran.SynchronizeTable("t1");
  tran.Insert<int,int>("t1",1,99);
string tempTable = "temp" + tran.ManagedThreadId+"_more";

//in case if previous process was interrupted and tempTable was not deleted
engine.Scheme.DeleteTable(tempTable);
```

```
   tran.Insert<int,int>(tempTable ,1,99);

 //do operations with temp table.....

  engine.Scheme.DeleteTable(tempTable);

   tran.Commit();
 }catch(System.Exception ex)
 {
       //ex handle
 engine.Scheme.DeleteTable(tempTable);
 }
 }
```

NOT ALLOWED:

```
using (var tran = engine.GetTransaction())
{
        tran.Insert<int,int>("t2",1,99);

        tran.SynchronizeTable("t1");

        tran.Insert<int,int>("t1",1,99);
        tran.Commit();
}
```

To synchronize tables by pattern we use special symbols:

* - all other symbols
# - all other symbols except slash, followed by slash and any other character
$ - all other symbols, except slash
tran.SynchronizeTable("Articles*") - will mean that we block for writing all tables which start from the word Articles, like:

 Articles123
 Articles231
etc.
Articles123/SubItems123/SubItems123
and so on.

tran.SynchronizeTable("Articles#/Items*") - will mean that we block for writing following tables, like:

Articles123/Items1257/IOo4564

but we don't block

Articles123/SubItems546

tran.SynchronizeTable("Articles**$**") will mean that we block for writing following tables, like:

Articles123
Articles456

and we don't block

Articles456/Items...

Slash can be effectively used for creating groups.

Sure we can combine patterns in one tran.SynchronizeTable command:

tran.SynchronizeTable("Articles1/Items$","Articles#/SubItems*",
"Price1","Price#/Categories#/El*")

Non-Unique Keys

In DBreeze tables all keys must unique.

But there are a lot of methods on how to store non-unique keys.

One of them is for every non-unique key to create a separate table and store all references to this key inside. Sometimes this approach is good.

But there is another useful approach.

Note, that DBreeze is a professional database for high performance and mission-critical applications. Developer spends a little bit more time on the Data Access Layer, but gets back very fast responses from the database.

 Imagine that you have plenty of Articles and every one has a price inside. You know that one of the requirements of your application is to show articles sorted by price. Another requirement is to show articles in the price range.

It can mean that except the table who holds articles you will need a special table where you

will store prices as keys, to be able to use DBreeze SelectForwardStartFrom or SelectForwardFromTo.

Developer, while inserting one article, has to fill two tables (it's a minimum for this example) Articles and Prices.

But how we can store prices as key - they are not unique.

Then we will make them unique.

```csharp
using DBreeze;
using DBreeze.Utils;
using DBreeze.DataTypes;

public class Article
{
        public Article()
        {
        Id = 0;
        Name = String.Empty;
        Price = 0f;
        }

        public uint Id { get; set; }
        public string Name { get; set; }
        public float Price { get; set; }
}

public void Example_NonUniqueKey()
{
        engine.Schema.DeleteTable("Articles");

        using (var tran = engine.GetTransaction())
        {
            uint id=0;


            Article art = new Article()
            {
                Name = "Notebook",
                Price = 100.0f
            };

            id++;
            tran.Insert<uint, DbMJSON<Article>>("Articles", id, art);

            byte[] idAsByte = id.To_4_bytes_array_BigEndian();
            byte[] priceKey =
 art.Price.To_4_bytes_array_BigEndian().Concat(idAsByte);
```

```csharp
                    Console.WriteLine("{0}; Id: {1}; IdByte[]: {2}; btPriceKey:
{3}", art.Name, id, idAsByte.ToBytesString(""), priceKey.ToBytesString(""));
                    tran.Insert<byte[], byte[]>("Prices", priceKey, null);

                    art = new Article()
                    {
                            Name = "Keyboard",
                            Price = 10.0f
                    };

                    id++;
                    tran.Insert<uint, DbMJSON<Article>>("Articles", id, art);

                    idAsByte = id.To_4_bytes_array_BigEndian();
                    priceKey =
art.Price.To_4_bytes_array_BigEndian().Concat(idAsByte);
                    Console.WriteLine("{0}; Id: {1}; IdByte[]: {2}; btPriceKey:
{3}", art.Name, id, idAsByte.ToBytesString(""), priceKey.ToBytesString(""));
                    tran.Insert<byte[], byte[]>("Prices", priceKey, null);


                    art = new Article()
                    {
                            Name = "Mouse",
                            Price = 10.0f
                    };

                    id++;
                    tran.Insert<uint, DbMJSON<Article>>("Articles", id, art);

                    idAsByte = id.To_4_bytes_array_BigEndian();
                    priceKey =
art.Price.To_4_bytes_array_BigEndian().Concat(idAsByte);
                    Console.WriteLine("{0}; Id: {1}; IdByte[]: {2}; btPriceKey:
{3}", art.Name, id, idAsByte.ToBytesString(""), priceKey.ToBytesString(""));
                    tran.Insert<byte[], byte[]>("Prices", priceKey, null);

                    art = new Article()
                    {
                            Name = "Monitor",
                            Price = 200.0f
                    };

                    id++;
                    tran.Insert<uint, DbMJSON<Article>>("Articles", id, art);

                    idAsByte = id.To_4_bytes_array_BigEndian();
                    priceKey =
art.Price.To_4_bytes_array_BigEndian().Concat(idAsByte);
                    Console.WriteLine("{0}; Id: {1}; IdByte[]: {2}; btPriceKey:
```

```csharp
{3}", art.Name, id, idAsByte.ToBytesString(""), priceKey.ToBytesString(""));
                tran.Insert<byte[], byte[]>("Prices", priceKey, null);

        //this article was added later and not reflected in the post
explanation
                art = new Article()
                {
                        Name = "MousePad",
                        Price = 3.0f
                };

                id++;
                tran.Insert<uint, DbMJSON<Article>>("Articles", id, art);

                idAsByte = id.To_4_bytes_array_BigEndian();
                priceKey =
art.Price.To_4_bytes_array_BigEndian().Concat(idAsByte);
                Console.WriteLine("{0}; Id: {1}; IdByte[]: {2}; btPriceKey:
{3}", art.Name, id, idAsByte.ToBytesString(""), priceKey.ToBytesString(""));
                tran.Insert<byte[], byte[]>("Prices", priceKey, null);

                tran.Commit();

        }


Console.WriteLine("*************************************************");

        //Fetching data >=
        using (var tran = engine.GetTransaction())
        {
        //We are interested here in Articles with the cost >= 10

                float price = 10f;
                uint fakeId = 0;

                byte[] searchKey =
price.To_4_bytes_array_BigEndian().Concat(fakeId.To_4_bytes_array_BigEndian());

                Article art=null;

                foreach (var row in tran.SelectForwardStartFrom<byte[],
byte[]>("Prices", searchKey, true))
                {
                        Console.WriteLine("Found key:
{0};",row.Key.ToBytesString(""));

                        var artRow = tran.Select<uint,
DbMJSON<Article>>("Articles", row.Key.Substring(4, 4).To_UInt32_BigEndian());
```

```csharp
                        if (artRow.Exists)
                        {
                        art = artRow.Value.Get;
                        Console.WriteLine("Articel: {0}; Price: {1}",
art.Name, art.Price);
                        }
                }
        }


Console.WriteLine("*************************************************");

        //Fetching data >
        using (var tran = engine.GetTransaction())
        {
        //We are interested here in Articles with the cost > 10

                float price = 10f;
                uint fakeId = UInt32.MaxValue;

                byte[] searchKey =
price.To_4_bytes_array_BigEndian().Concat(fakeId.To_4_bytes_array_BigEndian());

                Article art = null;

                foreach (var row in tran.SelectForwardStartFrom<byte[],
byte[]>("Prices", searchKey, true))
                {
                        Console.WriteLine("Found key: {0};",
row.Key.ToBytesString(""));

                        var artRow = tran.Select<uint,
DbMJSON<Article>>("Articles", row.Key.Substring(4, 4).To_UInt32_BigEndian());

                        if (artRow.Exists)
                        {
                        art = artRow.Value.Get;
                        Console.WriteLine("Articel: {0}; Price: {1}",
art.Name, art.Price);
                        }
                }

        }


}
```

Every article when is inserted to Articles table receives its unique id ot type uint:

Articles<uint,DbMJSON<Article>>("Articles")

You remember that in the namespace DBreeze.Utils there are a lot of extensions for converting different data types to byte[] and back. We can convert decimals, doubles, floats, integers etc. to byte[] and back.

Article price is float in our example and can be converted to byte[4] (sortable byte array from DBreeze.Utils, System.BitConverter will not give you such results).

As you see we had 4 articles, 2 of them had the same price.
We achieve uniqueness of the price on the byte level by concatenating two byte arrays.
First part is a price converted to byte array (for Article Keyboard):
float 10.0f -> AE-0F-42-40
Second part is uint Id from table Articles converted to byte array (for Article Keyboard):
uint 2 -> 00-00-00-02

when we concatenate both byte arrays for every article we will have such result:

Notebook; Id: 1; btPriceKey: AF-0F-42-40-00-00-00-01   //100f
Keyboard; Id: 2; btPriceKey: AE-0F-42-40-00-00-00-02   //10f
Mouse; Id: 3; btPriceKey: AE-0F-42-40-00-00-00-03   //10f
Monitor; Id: 4; btPriceKey: AF-1E-84-80-00-00-00-04   //200f

That's all exactly these final byte arrays we insert into table prices.

Now fetching data

Select Forward and Backward from table Prices will give you already sorted by price results.

More interesting is to get All prices starting from 10f.

For this we will use tran.SelectForwardStartFrom("Prices",btKey,true);

we need to get btKey.

We take our desirable 10f and convert to byte[]
float findPrice = 10f;
byte[] btKey = findPrice.To_4_bytes_array_BigEndian();

then we need to concatenate with the btKey full article id and here is a trick:

uint id = 0;
btKey = btKey.Concat(id.To_4_bytes_array_BigEndian())

will give us such btKey:

AE-0F-42-40-00-00-00-00

if we use it in tran.SelectForwardStartFrom("Prices",btKey,true);

we will receive all prices >= 10f.

If we

uint id = UInt32.MaxValue;
btKey = btKey.Concat(id.To_4_bytes_array_BigEndian())

will give us such btKey:
AE-0F-42-40-FF-FF-FF-FF

applying such key in tran.SelectForwardStartFrom("Prices",btKey,true);

we will receive a price only > 10f.

Sure when you got the key from value price (it's byte[]), you can make
row.Value.Substring(4,4).To_UInt32_BigEndian() - receive you uint id from table Articles and
retrieve value from table Articles by this key.

[20120521]

Fractal table structure.

Note, **investigation shows that it's not recommended to use this technique. To
distinguish data structures stored inside of one table use an initial byte in the key**.
E.g.
With the first byte 1 in the key we store entity "X":
 byte[] { 1 }.Concat(((long)1).ToBytes())
 byte[] { 1 }.Concat(((long)2).ToBytes())

byte[] { 1 }.Concat(((long)3).ToBytes())
With the first byte 2 in the key we store entity "Y":
 byte[] { 2 }.Concat(((long)1).ToBytes())
 byte[] { 2 }.Concat(((long)2).ToBytes())
 byte[] { 2 }.Concat(((long)3).ToBytes())
It can be a substitution of Nested / Fractal tables. Exactly such an approach is used in the object layer of DBreeze.

We call it with a fancy word "fractal", because it has a self-similar structure.

Actually, it's an ability to store in any kind of a value (of a Key/Value table) from 1 to N other tables + extra data. And in any kind of a nested table keys values other from 1 to N tables + extra data and so on, till you resources let you do that. Such a multi-dimensional storage concept.

It can also mean that in one value we can store objects of any complexity kind. Every property of this object which can be represented as a table (List or Dictionary) inherits all possibilities of the master table. We can again make favorite operations like Forward, Backward Skip, Remove, Add etc. and the same with sub-nested tables and sub-sub....-sub nested tables.

To insert a table in a value we need 64 bytes - it's a size of table root.

```
Table "t1"
 Key | Value
  1  | /...64 byte..../              /...64 byte..../                    /...64 byte..../
        Key<int>Value                Key<string>    Value
          1      /...64 byte..../       a5      /...64 byte..../ /...64 byte..../
          2      /...64 byte....//...64 byte..../  b6      string
          3                              t7      int
                               h8      long
  2  | /...64 byte..../
  3  | /...64 byte....//...64 byte..../ extra data /...64 byte..../ extra data /...64 byte..../
```

Note, it's not possible to copy the table which has values in nested tables with the techniques described before (simple bytes copying). But it is possible to automate this process, because the table root has a mark "dbreeze.tiesky.com" always starting at the same point from table root start, also the root length is fixed with 64 bytes, so one day we will make this recursive copy function.

Note, we are still thinking about the methods names which we use while fetching nested tables and we know that the time will place correct emphasis here also.

Fractal tables structure. Getting Started

Note, **investigation shows that it's not recommended to use this technique. To distinguish data structures stored inside of one table use an initial byte in the key**.
E.g.
With the first byte 1 in the key we store entity "X":
 byte[] { 1 }.Concat(((long)1).ToBytes())
 byte[] { 1 }.Concat(((long)2).ToBytes())
 byte[] { 1 }.Concat(((long)3).ToBytes())
With the first byte 2 in the key we store entity "Y":
 byte[] { 2 }.Concat(((long)1).ToBytes())
 byte[] { 2 }.Concat(((long)2).ToBytes())
 byte[] { 2 }.Concat(((long)3).ToBytes())
It can be a substitution of Nested / Fractal tables. Exactly such an approach is used in the object layer of DBreeze.

Every operation starts from the master table. Master table is a table which is stored in the Scheme and you perfectly know its name.

tran.Insert<int,string>("t1",1,"Hello");
tran.Insert<int,string>("t1/Points",1,"HelloAgain");

"t1" and "t1/Points" - are master tables.

So, let's assume we have a master table with the name "t1". Keys of this table are of integer type. Values can be different.

```
using (var tran = engine.GetTransaction())
    {
            tran.Insert<int, string>("t1", 1, "hello");
            tran.Insert<int, byte[]>("t1", 2, new byte[] { 1, 2, 3 });
            tran.Insert<int, decimal>("t1", 3, 324.34M);

    }
```

If you know what is stored under different keys you can always correctly fetch the values, on the lowest level they are always byte[] - byte array.

To insert a table we have designed new method

tran.InsertTable<int>("t1", 4, 0);

you need to supply one type for key resolving, value will be automatically resolved as byte array. As parameters you need to supply the master table name, key (4 in our example) and table index.

As you remember we can put more then 1 table in the value and every of it will reside 64 bytes.
So, if index = 0 then table will reside value bytes from 0-63, if index = 1 then table will reside value bytes from 64-127 etc....

In between you can put your own values, just remember not to overlap nested tables roots.

Again, we can say

tran.InsertTable<int>("t1", 4, 0);
tran.InsertPart<int, int>("t1", 4, 587, 64);

Key 4 will have 64 bytes of a table and then 4 reserved bytes for the value 587. You can work separately with them.

Note, method **InsertTable** gives us extra load telling us that we want to insert/change/modify. If the **table** didn't exist in that place it will be **automatically created**. Also Insert Table will notify the system that the thread, who is using it, tries to modify table "t1", that's why all necessary techniques like tran.SynchronizeTables, if you modify more than one master table, must be used. They are described in previous chapters.

We have another method

tran.SelectTable<int>("t1", 4, 0);

In contrast to InsertTable if a table is not found it will not be created.

Note, method **SelectTable** will not create a table if it doesn't exist and this method is recommended for **READING THREADS.** But also can be used by WRITING threads just to get the table without its creation.

Note, tran.**InsertTable** and **SelectTable** always return value of type **DBreeze.DataTypes.NestedTable**

NestedTable repeats by functionality Transaction class in the scope of table operations. You will find there all well known methods: Select SelectForward Backward, Insert, InsertPart, RemoveKey, RemoveAll etc.

First difference is that you don't need to supply the table name as a parameter.

Key   Value
1
2
3
4   /*....64 byte...table*/ /*4 bytes integer*/
        Key   Value
        1   Hi1
        2   Hi2
        3   Hi3

To build up such structure we do following code:

```
 tran
                .InsertTable<int>("t1", 4, 0)
                .Insert<int, string>(1, "Hi1")
                .Insert<int, string>(2, "Hi2")
                .Insert<int, string>(3, "Hi3");

tran.Commit();
```

This "functional programming" technique is possible due to returns of Insert - It returns the underlying NestedTable.

To read the data we do following:

```
 tran
                .SelectTable<int>("t1", 4, 0)
                .Select<int, string>(1)
                .PrintOut();
```

We will receive "Hi1"

PrintOut is a small "console out" helper for checking the content.

Lets iterate

```
 foreach (var row in tran
                                .SelectTable<int>("t1", 4, 0)
```

```
                        .SelectForward<int, string>()
        )
        {
        row.PrintOut();
        }
```

Note, if you try to Insert into a nested table after master-SelectTable you will receive an exception. Inserting (Removing, changing - etc all modifications) into all nested tables generations is allowed only starting from the master- InsertTable method.

Let's try more complex structure

```
Key   Value
1
2
3
4     /*....64 byte...table*/
      Key   Value
      1     Hi1
      2     /*....64 byte...table*/   /*....64 byte...table*/
            Key  Value            Key  Value
             1    Xi1              7    Piar7
             2    Xi2              8    Piar8

      3     Hi3
```

```
 var horizontal =
            tran
            .InsertTable<int>("t1", 4, 0);

            horizontal.Insert<int, string>(1, "Hi1");

            horizontal
            .GetTable<int>(2, 0)  //we use it to access next table generation
            .Insert(1, "Xi1")
            .Insert(2, "Xi2");
```

```
            horizontal
            .GetTable<int>(2, 1)
            .Insert(7, "Piar7")
            .Insert(8, "Piar8");

            horizontal.Insert<int, string>(3, "Hi1");

        //Here all values for all nested tables will be committed
        tran.Commit();
```

//Fetching value

```
            tran.SelectTable<int>("t1", 4, 0)
            .GetTable<int>(2, 1)
            .Select<int, string>(7)
            .PrintOut();
```

//Return will be "Piar7"

Note, there is no separate Commit or Rollback of the nested tables; they are done via master table Commit or Rollback.

[20120525]

Select returns DBreeze.DataTypes.Row

This **Row** we know from previous examples, but now it's enhanced with the new method GetTable(uint tableIndex), where you can get a nested table stored inside of this row by tableIndex. It works for master and for nested tables.

```
 using (var tran = engine.GetTransaction())
 {
 tran.InsertTable<int>("t1", 1, 1)
                 .Insert<uint, string>(1, "Test1")
```

```
                    .Insert<uint, string>(2, "Test2")
                    .Insert<uint, string>(3, "Test3");

            tran.Commit();


            //foreach (var row in tran.SelectTable("t1", 1, 1))  - also possible
   but...

            foreach (var row in tran.SelectForward<int,byte[]>("t1"))
            {
                    foreach (var r1 in row.GetTable(1).SelectForward<uint,
   string>())
                    {
                            r1.PrintOut();
                    }
            }
    }


//Result will be
1; "Test1"
2; "Test2"
3; "Test3"
```

InsertDictionary. SelectDictionary. InsertHashSet. SelectHashSet

We have created extra insert and select statements for master table and nested table to support direct casts of the DBreeze tables as a C# Dictionary and HashSet (list of unique keys).

```
  Dictionary<uint,string> _d=new Dictionary<uint,string>();
            _d.Add(10, "Hello, my friends");
            _d.Add(11, "Sehr gut!");

        Dictionary<uint, string> _b = null;

        using (var tran = engine.GetTransaction())
        {
        //Insert into Master Table Row
        tran.InsertDictionary<int, uint, string>("t1", 10, _d, 0,true);

        //Insert into Nested Table Dictionary
```

```
tran.InsertTable<int>("t1",15,0)
        .InsertDictionary<int, uint, string>(10, _d, 0,true);

tran.Commit();

//Select from master table
_b = tran.SelectDictionary<int, uint, string>("t1", 10, 0);

_b = tran.SelectTable<int>("t1",15,0)
        .SelectDictionary<int, uint, string>(10, 0);


}
```

tran.InsertDictionary<int, uint, string>("t1", 10, _d, 0,true);

will create following structure:

"t1"
Key<int>   Value<byte[]>
1
2
..
10   /*0-63 bytes new table*/
            Key<uint>   Value<string>
                10   "Hello, my friends"
                11   "Sehr gut!"

tran

    .InsertTable<int>("t1",15,0)
    .InsertDictionary<int, uint, string>(10, _d, 0,true);

will create following structure:

"t1"
Key<int>   Value<byte[]>
1
2
..
15   /*0-63 bytes new table*/
            Key<int>   Value<byte[]>
                ...   ...
                10   /*0-63 bytes new table*/
                            Key<uint>   Value<string>
                            10   "Hello, my friends"
                            11   "Sehr gut!"

Select will be used to get these values, Hashset has the same semantic.

Note, there is one important flag in InsertDictionary and InsertHashSet. It's the last parameter bool withValuesRemove.

If you supplied before Dictionary with keys 1,2,3....commit......then next time you supply Dictionary with values 2,3,4

if withValuesRemove = true
  then in db will stay keys 2,3,4
if withValuesRemove = false
  then in db will stay keys 1,2,3,4

These structures designed as help functions for:

- The quick method to store a set of keys/values into the nested tables from Dictionary or HashSet (InsertDictionary(....,false)).
- Help functions for small Dictionaries/HashSets to be stored and Selected with automatic removal and update (InsertDictionary(....,true)).

- Ability to get the full table of any Key/Value type as Dictionary or HashSet - right in memory.

[20120526]

We have also added Insert/Select Dictionary/HashSet for the tables themselves (not just moved by levels)

We can make following:

inserting right into t1 table values represented as Dictionary:
             tran.InsertDictionary<int, int>("t1", new Dictionary<int, int>(), false);

inserting into t1 row 1 a table which locates from 0 byte of row a Dictionary:

             tran.InsertTable<int>("t1", 1, 0).InsertDictionary<uint, uint>(new Dictionary<uint, uint>(), false);

Corresponding selects:
tran.SelectDictionary<int, int>("t1");
tran.SelectTable<int>("t1", 1, 0).SelectDictionary<uint, uint>();

The same for HashSets.

[20120529]

Nested tables memory management.

Note, **investigation shows that it's not recommended to use this technique. To distinguish data structures stored inside of one table use an initial byte in the key**.
E.g.
With the first byte 1 in the key we store entity "X":
 byte[] { 1 }.Concat(((long)1).ToBytes())
 byte[] { 1 }.Concat(((long)2).ToBytes())
 byte[] { 1 }.Concat(((long)3).ToBytes())
With the first byte 2 in the key we store entity "Y":
 byte[] { 2 }.Concat(((long)1).ToBytes())
 byte[] { 2 }.Concat(((long)2).ToBytes())
 byte[] { 2 }.Concat(((long)3).ToBytes())
It can be a substitution of Nested / Fractal tables. Exactly such an approach is used in the object layer of DBreeze.

We have a situation of memory growth in case we use lots of nested tables inside of one transaction. Support of a table takes a memory amount.

Master table and nested into it tables share the same physical file. Current engine automatically disposes of the master table and all nested tables when the transaction (working with master table) is finished. But only in cases when parallel threads don't read from the same table at the same time. Master table and nested into it tables will be disposed together with the last working with this table transaction. If we write into the table once per 7 seconds and read once per 2 seconds, definitely this table will be able to free residing memory in-between.

Some more situations. For example we insert data in such manner:

```
using (var tran = engine.GetTransaction())
{
        for (int i = 0; i < 100000; i++)
        {
                tran.InsertTable<int>("t1", i, 1)
                    .Insert<uint, uint>(1, 1);
        }

        tran.Commit();
}
```

Really bad case for the memory. In this case we have to open 100000+1(master) tables and hold them in memory till tran.Commit();

In our tests used memory has grown up from 30MB (basic run of a test program) up to 350MB...after transaction was finished the process size didn't change, but those 320MB were marked to be collected by .NET Garbage Collector, so calling GC.Collect (or using the process further) brings back to 30MB.

And for now it's hard to find out how to avoid this memory growth. It's not so critical when you insert in small chunks (100 records). So you must remember that.

Another case:

Looks even more interesting. When we select data

```
using (var tran = engine.GetTransaction())
{
        for (int i = 0; i < 100000; i++)
        {
                var row = tran.SelectTable<int>("t1", i, 1)
                    .Select<uint, uint>(1, 1);
```

```
            if(row.Exists)
            {
                    //..do
            }
        }
}
```

Here, after every loop iteration we don't need any more used tables, but it still stays in memory and makes it grow. In this example memory has grown up from 30MB up to 135MB, sure if you select more records it will need more memory resource.

Exactly for such a case we had to integrate **table.Close** method.

To use Close, we need a variable for accessing this table. Our code will look like this now:

```
using (var tran = engine.GetTransaction())
{


        foreach (var row in tran.SelectForward<int, byte[]>("t1"))
        {
            var tbl = row.GetTable(1);

            if (!tbl.Select<uint, uint>(1).Exists)
            {
                    Console.WriteLine("not");
            }

            tbl.CloseTable();


        }
}
```

Now memory holds the "necessary level".

Note, When we call the NestedTable.Close method, we want to close the current table and all nested tables in it. Every master-table InsertTable or SelectTable (and nestedTable.GetTable) increase "open quantity" variable by 1, every CloseTable decreases value by 1, when value is less than 1, then the table with all nested in it tables will be closed.
If we forget to close the table then it will be open till all operations with the master table are finished and automatic disposal works.

Note, NestedTable.Dispose calls CloseTable automatically,

so we can make:

```
using (var tran = engine.GetTransaction())
{
        using(var tbl = row.GetTable(1))
        {
                if (!tbl.Select<uint, uint>(1).Exists)
                {
                        Console.WriteLine("not");
                }
        }
}
```

Rules.

- Don't close the table before you Commit or Rollback it.
- Transaction end will close master and nested tables automatically if no other threads are working with it, probably parallel thread will close it after finish.
- Close table instances manually if operations with the table are very intensive and there is no chance that it will be closed automatically.
- Control InsertTable the same way as SelectTable.
- It's possible to close tables of all nesting generations, depending upon your table structure. They will be closed starting from called generation.

This chapter is on the level of the experiment.

Secondary Index. Direct key select.

Here we present another experimental approach.

If we need to support other indices then our table key, where we store our objects we need to create other tables where keys will be secondary index etc. In the secondary index table we can store a direct pointer to the first table with the object in contrast with the key.

When we insert or change the key we have an ability to obtain its file pointer:

```
byte[] ptr =null;

using (var tran = engine.GetTransaction())
{

                tran.Insert<int, int>("t1", 12, 17, out ptr);
```

```
            tran.SelectDirect<int, int>("t1", ptr).PrintOut();

            tran.ChangeKey<int>("t1", 12, 15,out ptr);

            tran.SelectDirect<int, int>("t1", ptr).PrintOut();
}
```

then we can get the value by pointer economizing time for the search of the first table key.

Note, when we update the primary-table, which holds full information about the object, its pointer can be moved, that's why our DAL must update the value (pointer to the primary table key) in the secondary table also. When we delete from the primary table, we must delete the same transaction from the secondary index table also.

The same we can make inside of nested tables.

Note, for nested tables SelectDirect must be used exactly from the table where you are searching information to avoid collisions:

```
byte[] ptr =null;

using(var tbl = tran.InsertTable<int>("t3", 15, 0))
{
    tbl.Insert<int, int>(12, 17, out ptr);
    tran.Commit();
}

using(var tbl = tran.SelectTable<int>("t3", 15, 0))
{
var row = tbl.SelectDirect<int, int>(ptr);
        row.PrintOut();

}
```

Note, we can get pointer to the value inside of **Insert**, **InsertPart** and **ChangeKey** for primary and nested tables.

[20120601]

Dynamic-length data blocks and binding them to Row.Column.

Inside of the table we have keys and values. If to think about the value as row with columns, that gives us ability to store in one row independent data types, which we can access using Row.GetValuePart(uint startIndex, uint length) and everything seems to be good, when our

data types have fixed length. But sometimes we need to store inside of columns dynamic-length data structures.

For this we have developed following method inside of the transaction class:

public byte[] InsertDataBlock(string tableName, byte[] initialPointer, byte[] data)

Data blocks live in parallel with the table itself and inherit the same data visibility behavior for different threads like other structures.

Nested tables also have an InsertDataBlock method.
Note, InsertDataBlock always returns a byte[] of the same length - 16 bytes - it's a definition of the stored value, because returned value length is fixed we can use it as a column inside of a Row.

Note, if 2 parameter initialPointer is NULL then a new data block will be created for the table, if not NULL it can mean that such a data block already exists and DBreeze will try to overwrite it.
Note, data-blocks obey transaction rules, so till you commit "updated" data-block, parallel reading threads will continue to see its last-committed value. We can also rollback changes.

After we insert data-block we want to store its pointer inside of a row, to have an ability to get it later:

byte[] dataBlockPtr = tran.InsertDataBlock("t1", null, new byte[] { 1, 2, 3 });

here we have received data-block pointer and we want to store this pointer in t1 row

tran.InsertPart<int, byte[]>("t1", 17, dataBlock, 10);

We have stored a pointer to the data-block inside of "t1" key (17) starting from index 10, the pointer has always fixed length 16 byte, starting from index 26 we can go on to store other values.

Now we want to retrieve the data back:

It's possible via Row object:

> var row = tran.Select<int, byte[]>("t1", 17);
> byte[] res = row.**GetDataBlock**(10);

Note, Data-Block can store null value.

Updated:
Also, we can now directly get DataBlocks from transaction:

//When datablock is saved in master table
tran.SelectDataBlock("t1",dataBlockPointer);
//When datablock is saved in nested table
tran.SelectTable<int>("t1",1,0).SelectDataBlock(dataBlockPointer)

If we want to store link to the data-block inside of nested table row, we must make it via Nested Table method:

```
var tbl = tran.InsertTable<int>("t1", 18, 0);

byte[] dbp = tbl.InsertDataBlock(null, new byte[] { 1, 2, 3 });

tbl.InsertPart<int, byte[]>(19, dbp, 10);

tran.Commit();

tbl.CloseTable();


tbl = tran.SelectTable<int>("t1", 18, 0);
var row = tbl.Select<int, byte[]>(19);
byte[] fr = row.GetDataBlock(10);

if (fr == null)
Console.WriteLine("T1 NULL");
else
Console.WriteLine("T1 " + fr.ToBytesString());
```

System understands empty pointers to the data-block. In following example we try to get not-existing data-block, then update it and write pointer back:

```
var row = tran.Select<int, byte[]>("t1", 17);

byte[] dataBlock = row.GetDataBlock(10);

dataBlock = tran.InsertDataBlock("t1", dataBlock, new byte[] { 1, 2, 3, 7, 8 });

tran.InsertPart<int, byte[]>("t1", 17, dataBlock, 10);

tran.Commit();
```

Hash Functions of common usage. Fast access to long strings and byte arrays.

DBreeze search-trie is a variation of radix trie implementation optimized by many parameters - © **Liana-Trie**. So, if we have keys of type int (4 bytes), we will need from 1 up to 4 HDD hits to get a random key (we don't talk about HDD possible problems and OS file system fragmentations here). If we have keys of type long (8 bytes) we will need from 1 up to 8 hits, depending upon keys quantity and character. If we store longer byte arrays, we will need from 1 up to max-length of the biggest key hits. If we store in one table 4 such string keys:

key1: http://google.com/hi
key2: http://google.com/bye
key3: http://dbreeze.tiesky.com
key4: abrakadabra

to get randomly key1 we will need http://google.com/h - 19 hits
to get randomly key2 we will need http://google.com/b - 19 hits
to get randomly key3 we will need http://d - 8 hits
to get randomly key4 we will need only 1 hit

(after you find a key in range selects, searching of others, inside of iteration will work fast)

So, if we need to use StartsWith, or we need sorting of such tables, we have to store keys like they are.
But if we need just random access to such keys, the best approach will be to store not the full keys but only their 4/8 or 16 bytes HASH-CODES. Also, hashed keys and values with direct physical pointers, can represent secondary indexes. For example, in the first table we store keys, like they are, with the content and in the second table we store hashes of those keys and physical pointers to the first table. Now we can get a sorted view and have fastest random access (from 1 up to 8 hits, if hash is of 8 bytes).

Hashes can have collisions. We have integrated into DBreeze sources MurMurHash3 algorithm (which returns back 4 bytes hash) and added two more functions to get 8 bytes and 16 bytes hash code. We recommend using those 8 bytes or 16 bytes functions to stay collision-safe with a very high probability. If you need a 1000% guarantee, use a nested table under every hash and store in it real key (or keys in case of collisions), for checking or some kind of other technique, like serialized list of keys with the same hash code.

DBreeze.Utils.Hash.MurMurHash.MixedMurMurHash3_64 - 8 byte - returns ulong
and
DBreeze.Utils.Hash.MurMurHash.MixedMurMurHash3_128 - 16 byte - return byte[]

[20120628]

Row has the property LinkToValue (actually it's a link to Key/Value), for getting a direct link to the row and using it together with SelectDirect. All links (pointers to key/value pairs) now return fixed 8 bytes and can be stored as virtual columns in rows.

Also, we can now directly get DataBlocks from transaction:
//When datablock is saved in master table
tran.SelectDataBlock("t1",dataBlockPointer);
//When datablock is saved in nested table
tran.SelectTable<int>("t1",1,0).SelectDataBlock(dataBlockPointer)

[20120905]

Integrated incremental database backup ability.

To make it working instantiate dbreeze like this:

```
DBreezeConfiguration conf = new DBreezeConfiguration()
                {

DBreezeDataFolderName = @"D:\temp\DBreezeTest\DBR1",

                Backup = new Backup()
                {
                BackupFolderName = @"D:\temp\DBreezeTest\DBR1\Bup",
                IncrementalBackupFileIntervalMin = 30
                }

            };

            engine = new DBreezeEngine(conf);
```

If the Backup object is not included in configuration or DBreeze is instantiated without configuration, like it was before, incremental backup will be switched off. Sure, there is still a DBreeze constructor without configuration parameters at all.

If you have existing databases you can make a full copy ("snapshot") and start to continue to work with the incremental backup option switched on. Backup will create once per "IncrementalBackupFileIntervalMin " a new file (old files are released and can be copied out and deleted). Current backup file is always locked by dbreeze. You have to specify a folder for dbreeze incremental backup files "BackupFolderName". That's all.

If you start a new database with incremental backup option, then later you will be able to recreate the whole db from backup files, if you have started from a "snapshot" then backup files can bring your "snapshot" to current db state.

You can restore backup in the folder where your snapshot resides or, if incremental backup was switched on from the beginning, into the empty folder.

Example of backup restoration is shown in VisualTester - satellite project to DBreeze solution, under button "RestoreBackup".

Switched on incremental backup option brings Write speed decrease, Read speed is untouched.

Inserting one million of integers without backup option - 9 sec with option - 17 sec.

[20120922]

!!!!!!!!!!!!!!!!!!!!!!!!! IMPORTANT for those WHO USED DATABASE TILL THIS PERIOD !!!!!!!!!!!!!!!!!!!!!!!!! AND USED DBreeze.Utils ByteProcessing extensions: DateTime !!!!!!!!!!!!!!!!!!!!!!!!!.To_8_bytes_array_BigEndian(); and !!!!!!!!!!!!!!!!!!!!!!!!! byte[] To_DateTime_BigEndian();

After attaching new DBreeze and recompilation of the project you will see errors, because such functions don't exist any more in DBreeze.

Why?
It's an issue, a historical issue. Our DBreeze generic type converter (we use it in tran.Insert<DateTime,DateTime .. tran.InsertPart<DateTime etc.) was written before some ByteProcessingUtils functions and somehow DateTime was converted first to **ulong** and then to byte[]. Otherwise, To_DateTime_BigEndian() and To_8_bytes_array_BigEndian() from DBreeze.Utils used **long**, such unpleasant things.

Well, now what?
So, we have decided to leave DateTime converter to work with ulong. It doesn't have an influence on the speed, and we don't need to recreate many existing databases.

We have created instead such functions in DBreeze.Utils.ByteProcessing: **public static DateTime To_DateTime(this byte[] value)** and this will work with **ulong** and **public static byte[] To_8_bytes_array(this DateTime value)** which recreates DateTime from 8-byte array. **With these functions we recommend working in the future**. The same algorithms are used by generic converters.

But, if you have already used manual DateTime conversions, we have left two functions for compatibility:

public static byte[] **To_8_bytes_array_zCompatibility**(this DateTime value)
 (this you must put in the code instead of old To_8_bytes_array_BigEndian concerning DateTime) and

DateTime **To_DateTime_zCompatibility**(this byte[] value) (this you can use instead of old To_DateTime_BigEndian)

They both go on to work with DateTime as long to byte[].

So, think about that and do what you should do :)
Actually, nothing should stop us in the light way of God's Love!

Storing in the value columns of the fixed size.

For the last few months we have created many tables with different value configurations, combining ways of data storage. One of the most popular ways is handling value byte[] as a set of columns of fixed length. We found out that we have lack of null-able data types and for this we have added in DBreeze.Utils.ByteProcessing a range of extensions for all standard null-able data types:

You take any standard null-able data type int?, bool?, DateTime?, decimal?, float? uint? etc. and convert it into byte[] using DBreeze.Utils extensions:

 public static byte[] To_5_bytes_array_BigEndian(this int? value)
or
public static byte[] To_16_bytes_array_BigEndian(this decimal? input)
etc...

and the same backward:

public static DateTime? To_DateTime_NULL(this byte[] value)
or
public static ushort? To_UInt16_BigEndian_NULL(this byte[] value)
…
etc. with NULL in the end

Note, that practically all null-able converters create byte[] on 1 byte longer than not null-able.

Sometimes in one value we hold some columns of fixed length then some DataBlocks, which represent pictures or so and then DataBlocks which represent big-text or json - serialized object parts. But we found out that we miss storing text in this way, like standard RDBMS make that: nvarchar(50) NULL or varchar(75). Sure we can use DataBlocks for that, but sometimes we don't want it, especially having that DataBlock reference will reside 16 bytes.

We have added in DBreeze.Utils ByteProcessing two more extensions:

public static byte[] To_FixedSizeColumn(this string value, short fixedSize, bool

isASCII)

and

public static string From_FixedSizeColumn(this byte[] value, bool isASCII)

They both will emulate behavior of RDBMS text fields of the fixed reservation length. Maximum 32KB. Minimum 1 byte for ASCII text and 4 bytes for UTF-8 text.

Take a string (it can be also NULL) and say:

string a = "my text";
byte[] bta = a.**To_FixedSizeColumn(50,true);**

and you will receive a byte array of 50+2 = **52 bytes** that you can store in your value from a specific place (let's say 10).

Note, returned size will always be 2 bytes longer. We need them to store the length of the real text inside of the fixed-size array and NULL flag.

Then take your value.Substring(10,52).**From_FixedSizeColumn(true)** and you will receive your "my text". isASCII must be set to false if you store UTF-8 values. If size of the text exceeds the fixedSize parameter, then value will be truncated (correct algorithm is used, so only full UTF-8 chars will be stored without any garbage bytes in the end).

Sometimes, it's very useful as a first byte of the value to set up a row version, then, depending upon this version, the further content of the value can have different configurations of the content.

[20121012]

Behavior of the iterators with the modification instructions inside.

Let's assume that before every following example, we delete table "t1" and then execute such insert:

```
using (var tran = engine.GetTransaction())
{

        for (int i = -200000; i < 800000; i++)
        {
            tran.Insert<int, int>("t1", i, i);
        }

        tran.Commit();
}
```

Sometimes it's interesting for us to make table modifications while iteration, like here:

```
using (var tran = engine.GetTransaction())
{
      //t1 is not in modification list, enumerators visibility scope is
"parallel read"

foreach (var row in tran.SelectForward<int, int>("t1"))
        {
                tran.RemoveKey<int>("t1", row.Key);
        }

tran.Commit();
}
```

In such an example it will work well.

In the next example it will also work:


```
using (var tran = engine.GetTransaction())
{

      tran.SynchronizeTables("t1");

//t1 is in modification list, enumerators visibility scope is "synchronized
read/write"
//probably we can see changes made inside of iteration procedure.


      var en = tran.SelectForward<int, int>("t1").GetEnumerator();

      while (en.MoveNext())
        {
                tran.RemoveKey<int>("t1", en.Current.Key);
```

```
            }

    tran.Commit();
    }
```

Enumerator en, refers to writing root at this moment, because our table was added into the modification list (by SynchronizeTable or any other modification command, like insert, remove etc...), and changes of the table, even before committing, can be reflected inside the enumerator.

But, we delete the same key which we read, that's why this task will be accomplished well. We don't insert or delete "elements of the future iterations".

In the next example we can have undesired behavior:

```
using (var tran = engine.GetTransaction())
{

        tran.SynchronizeTables("t1");

//t1 is in modification list, enumerators visibility scope is "synchronized
read/write"
//probably we can see changes made inside of the iteration procedure.

int pq = 799999;

        var en = tran.SelectForward<int, int>("t1").GetEnumerator();

        while (en.MoveNext())
            {
                    tran.RemoveKey<int>("t1", pq);
             pq--;
            }

    tran.Commit();
    }
```

We will not delete all keys in the previous example. Enumerators will stop iterating somewhere in the middle, where exactly - depends upon key structure and not really useful for us.

So, if you are going to iterate something and change possible "elements of the future iterations", there is no guarantee for the correct logic execution. This concerns synchronized iterators.

To make it correct, we have added for every range select function an overload with the parameter **bool AsReadVisibilityScope**. It concerns nested tables range select functions

also.

Now we can make something like this:

```
using (var tran = engine.GetTransaction())
{

        tran.SynchronizeTables("t1");

//t1 is in modification list, enumerators visibility scope is "synchronized
read/write"
//probably we can see changes made inside of the iteration procedure.

int pq = 799999;

        var en = tran.SelectForward<int, int>("t1", true).GetEnumerator();

        while (en.MoveNext())
            {
                    tran.RemoveKey<int>("t1", pq);
             pq--;
            }

tran.Commit();
}
```

All keys will be deleted correctly. Because our enumerator's visibility scope will be the same as in parallel thread, it will see only committed data projection, before the start of the current transaction.

Now we can vary which visibility scope for the enumerator, whose table is inside of the modification list, we want to choose, synchronized or parallel. Default range selects, without extra parameters, if table is in modification list will return synchronized view.

[20121015]

Secondary Indexes. Going deeper.

Transaction/NestedTable method **Select** now is also overloaded with **bool AsReadVisibilityScope,** for the same purposes as described in the previous chapter.

Let's assume that we have an object:

```
public class Article
{
```

```
        [PrimaryKey]
        public long Id = 12;

        public string Name = "A1";

        [SecondaryKey]
        public float Price = 15f;
 }
```

Primary and Secondary keys attributes, for now, don't exist in DBreeze. But the idea is the following: from field "Id" we want to make Primary index/key and from field "Price" we want to create one of our secondary indexes.

For now DBreeze doesn't have extra object layer, so we would make such save in the following format:

```
 using DBreeze;
 using DBreeze.Utils;

 public void SaveObject(Article a)
 {
        byte[] ptr=null;
         using (var tran = engine.GetTransaction())
              {
//Inserting into Primary Table
              tran.Insert<long,byte[]>
("Article",
a.Id.To_8_bytes_array_BigEndian(),                    //Id - primary key
a.Name.To_FixedSizeColumn(50, false)          //let it be not DataBlock
.Concat(
a.Price.To_4_bytes_array_BigEndian()
),
out ptr                               //getting back a physical pointer
);

//Inserting into Secondary Index table

              tran.Insert<byte[],byte[]>
("ArticleIndexPrice",
a.Price.To_4_bytes_array_BigEndian()     //compound key: price+Id
.Concat(
     a.Id.To_8_bytes_array_BigEndian()
,
ptr           //value is a pointer to the primary table
);
)

              tran.Commit();
 }
```

```
    }
```

Something like this. In real life all primary and secondary indexes could be packed into the nested tables of one MasterTable under different keys.

We have filled 2 tables. First is "Article". As key there we store Article.Id as value we store article name and price. Second table is "ArticleIndexPrice". Its key is constructed from (float)Price+(long)ArticleId - it's unique, sortable, comparable and searchable. Such a technique was described in previous articles. As a value we store a physical pointer to the primary key inside of the "Article" table. When we have such a physical pointer, searching for Key/Value of the PrimaryTable "Article" is only one HDD hit.

But keys and values are not always static. Sometimes we remove articles, sometimes we change the price or even expand the value (in the last case, we need to save a new physical pointer into a secondary index table).

If we remove an Article, we must remove the compound key from the table "ArticleIndexPrice" also.
When we update the price, inside of table Article, we must delete the old compound key from the table "ArticleIndexPrice" and create a new one.

It means that every time when we insert something into table Article - it can be counted as a probable update, and we must check if a row with such Id exists before insert. If yes then we must read it, delete the compound key, construct and insert a new compound key into the table "ArticleIndexPrice" and finally update the value in the table "Article".

This all can slow down the insert process very much.

That's why we have added for every modification command, inside of the transaction class and nested table class, useful overloads:

Modification commands overloads (the same for nested tables):

public void **Insert**<TKey, TValue>(string tableName, TKey key, TValue value, out byte[] refToInsertedValue, **out bool WasUpdated**)

public void **InsertPart**<TKey, TValue>(string tableName, TKey key, TValue value, uint startIndex, out byte[] refToInsertedValue, **out bool WasUpdated**)
public void **ChangeKey**<TKey>(string tableName, TKey oldKey, TKey newKey, out byte[] ptrToNewKey, **out bool WasChanged**)

public void **RemoveKey**<TKey>(string tableName, TKey key, **out bool WasRemoved**)

Actually, Dbreeze, when inserting data, knows if it's going to be an update or a new insert.

That's why Dbreeze can notify us about this.

We go on to insert data in the usual manner. If flag **WasUpdated** equals to true, then we know that it was an update. We can use our new, **overloaded** with visibility scope parameter, **Select** to get the key/value pair, which was before modification and change the secondary index table. We need to make this action only in case of update/remove/change command, but not in case of the new insert.

[20121016]

Secondary Indexes. Going deeper. Part 2

If we store inside of value DataBlocks (not just serialized value or columns of fixed length), before we make an update of such value, we must read it in any case previous value content (to get DataBlocks initial pointers for updates). So, again every insert can be counted as probable updates. Following technique/benchmark shows us time consumption for reading previous row value version before insert:

This is a standard insert:

```
using (var tran = engine.GetTransaction())
            {
            byte[] ptr=null;
            bool wasUpdated = false;
            DBreeze.Diagnostic.SpeedStatistic.StartCounter("a");
            for (int i = -200000; i < 800000; i++)
            {
                        tran.Insert<int, int>("t1", i, i, out ptr, out
wasUpdated);
            }
DBreeze.Diagnostic.SpeedStatistic.PrintOut("a", true);
            tran.Commit();
            }
```

Operation took **1.5 sec (year 2015 HDD)**. **1 MLN** of **inserts**.

This is an insert with getting previous row version before insert:

```
using (var tran = engine.GetTransaction())
{
        byte[] ptr=null;
        DBreeze.DataTypes.Row<int, int> row = null;

        DBreeze.Diagnostic.SpeedStatistic.StartCounter("a");
```

```
        for (int i = -200000; i < 800000; i++)
        {
```

//Note, we use Select with VisibilityScope=Parallel Read

```
            row = tran.Select<int, int>("t1", i,true);

            if (row.Exists)
            {
                //do update
                tran.Insert<int, int>("t1", i, i, out ptr);
            }
            else
            {
                //do insert
                tran.Insert<int, int>("t1", i, i, out ptr);
            }
        }
        DBreeze.Diagnostic.SpeedStatistic.PrintOut("a", true);

        tran.Commit();
}
```

This operation took **10600 ms (10 sec ? old benchmark from year 2012)**. **1 MLN** of **inserts,** distinguishing between updates and inserts.
Remember, that DBreeze insert and select algorithms work with maximum efficiency in bulk operations, when keys are supplied sorted in ascending order (descending is a bit slower). So, sort bulk chunks in memory before inserts/selects.

Previous 2 examples were about pure inserts, and we run them again having data already in the table, so all records have to be updated:

1 example - 1 MLN of updates took 28 sec
2 example - 1 MLN of updates with (getting row previous version) took 36 sec.

[20121023]

DBreeze like an in-memory database.

Dbreeze can also reside fully in-memory. It's just a feature. Having the same functionality as the disk-based version.

Instantiating example:

```csharp
DBreeze.DBreezeEngine memeng = new DBreezeEngine(new DBreezeConfiguration()
            {
                        Storage =  DBreezeConfiguration.eStorage.MEMORY
            });


using (var tran = memeng.GetTransaction())
 {

            for (int i = 0; i < 1000000; i++)
            {
                    tran.Insert<int, int>("t1", i, i);
            }

            Console.WriteLine(tran.Count("t1"));

            tran.Commit();

 }
```

It works **a bit** slower on insert than .NET Dictionary or SortedDictionary, because it has lots of sub-system specific elements inside, designed to work with very large data sets, without index fragmentation after continuous inserts, updates and deletes.
But it gives us the ability of a very powerful search, without full scan of the data in comparison with .Net Dictionaries, when we talk about not just existing key picking.

"Out-of-the-box" bulk insert speed increase.

We have increased the standard bulk insert speed of DBreeze (about 5 times), by adding a special memory cache layer before flushing data on the disk. By standard configuration, 20 tables, which are written in parallel, receive such memory buffers of size 1MB each, before disk flush. The 21-th (and so on, parallel) will write without a buffer. After disposing of the writing transactions other tables can receive such a buffer, so it's not bound to the tables names - tables are chosen automatically right in time of the insert.

Now DBreeze, in standard configuration, can store in bulk (ascending ordered) 500K records per 1 seconds (Benchmark PC is taken). 6 parallel threads could write into 6 different tables 1MLN of records each, for the 3.4 seconds, which was about 40MB/s and 1.7 MLN simple records per second (see Benchmarking document).

[20121101]

Iterations SelectBackwardStartsWithClosestToPrefix and SelectForwardStartsWithClosestToPrefix.

They both concern master and nested tables.

If we have in the table string keys:

"check"
"sam"
"slash"
"slam"
"what"
string prefix = "slap";

```
foreach (var row in tran.SelectForwardStartsWithClosestToPrefix<string, byte>("t1", prefix))
{
        Console.WriteLine(row.Key);
}
```

Result:

slam
slash

and for
```
foreach (var row in tran.SelectBackwardStartsWithClosestToPrefix<string, byte>("t1", prefix))
```


Result:

slash
slam

[20121111]

Alternative tables storage locations.

Starting from the current DBreeze version we are able to set up table locations by table names patterns globally. We can mix tables' physical locations inside of one DBreeze instance. Tables can reside in different folders, on different hard drives and even in memory.

DBreezeConfiguration object is enriched with the public accessible Dictionary

AlternativeTablesLocations.

Now we can create DBreeze configuration in the following format:

```
DBreezeConfiguration conf = new DBreezeConfiguration()
            {
            DBreezeDataFolderName = @"D:\temp\DBreezeTest\DBR1",
            Storage = DBreezeConfiguration.eStorage.DISK,
            };
```

//SETTING UP ALTERNATIVE FOLDER FOR TABLE t11
conf.AlternativeTablesLocations.Add("t11",@"D:\temp\DBreezeTest\DBR1\INT");

//SETTING UP THAT ALL TABLES STARTING FROM "mem_" must reside in-memory
conf.AlternativeTablesLocations.Add("mem_*", **String.Empty**);

//SETTING UP Table pattern to reside in different folder
conf.AlternativeTablesLocations.Add("t#/Items", @"D:\temp\DBreezeTest\DBR1\EXTRA");

 engine = new DBreezeEngine(conf);

So, if the value of the Dictionary AlternativeTablesLocations key is **empty**, the table will be automatically forced to work **in-memory**. If pattern for the table is not found, table will be created, overriding DBreeze main configuration settings (DBreezeDataFolderName and StorageType).

If **one table** corresponds **to some patterns**, the **first** one will be **taken**.

Patterns logic is the same as in "Transaction Synchronize Tables":

$ * # - pattern extra symbols
"U" - intersects, "!U" - doesn't intersect

* - 1 or more of any symbol kind (every symbol after * will be cutted): Items* U Items123/Pictures etc...
# - symbols (except slash) followed by slash and minimum another symbol: Items#/Picture U Items123/Picture
$ - 1 or more symbols except slash (every symbol after $ will be cutted): Items$ U Items123; Items$!U Items123/Pictures

Patterns can be combined:

Items#/Pictures#/Thumbs* can intersect *Items1/Pictures125/Thumbs44* or *Items458/Pictures4658/Thumbs1000* etc...

Incremental backup restorer works on the file level and knows nothing about the user's

logical table names. It will restore all tables in one specified folder. Later, after starting DBreeze and reading the scheme, it's possible manually to reside disk table files into corresponding physical places due to the storage logic.

[20130529]

Speeding up batch modifications (updates, random inserts)

To economize disk space DBreeze tries to utilize the same HDD space, if it's possible, in case of different types of updates.
There are 3 places where updates are possible:
- Update of search trie nodes (LianaTrie nodes)
- Update of Key/Values
- Update of DataBlocks

To be sure that overwriting data files will not be corrupted in case of power loss, first we have to write data into a rollback file, then into a data file. DBreeze in standard mode excludes any OS intermediate cache (only internal DBreeze cache) and makes writes to the "bare metal". Today's HDDs and even SSDs are quite slow for random writes. That's why we use a technique of changing random writes into sequential writes.

When we use DBreeze, for standard data accumulation of the random data from different sources, inside of small transactions, the speed degradation is not so visible. But we can see it very well when we need to update a batch of specific data.

We DON'T SEE SPEED DEGRADE, when we insert a batch of growing up keys - any newly inserted key is always bigger than maximal existing key (SelectForward will return newly inserted key as the last one). For such a case we should do nothing.

We CAN SEE SPEED DEGRADE, when we update a batch of values or data-blocks or if we insert a batch of keys in random order and, especially, if these keys have high entropy.

For such cases we have integrated new methods for transactions and for nested tables:

tran.**Technical_SetTable_OverwriteIsNotAllowed("t1");**

or

var tblABC = tran.InsertTable<byte[]>("masterTable", new byte[] { 1 }, 0);
tblABC .**Technical_SetTable_OverwriteIsNotAllowed();**

- This technique is interesting for the transactions with specific batch modifications, where speed really matters. Only developers can answer this question and find a balance.
- This technique is not interesting for the memory based data stores.
- These methods work only inside of one transaction and must be called for every table or nested table separately, before the table modification command.
- When a new transaction starts, overwriting automatically will be allowed again for all tables and nested tables.
- Overwriting concerns all: search trie nodes, values and data blocks.
- Remember always to sort the batch ascending by key, before insert - it will economize HDD space.

Of course this technique makes the data file bigger, but it returns the desired speed. All data which could be overwritten will be written to the end of the file.

Note

When **Technical_SetTable_OverwriteIsNotAllowed** is used, **InsertPart** still tries to update values that can lead to speed loss. If we need the speed while update, we can use such **workaround**:
- don't use InsertPart, only Insert
- read the whole value into memory as byte[]
- then change its middle part (with DBreeze.Utils.BytesProcessing CopyInside or CopyInsideArrayCanGrow)
- insert the complete value.
- All the time **Technical_SetTable_OverwriteIsNotAllowed** can be on.

Source code received a new folder DBreeze\bin\Release\NET40 where we store DBreeze.dll ready for MONO and .NET4> usage. This folder DBreeze\bin\Release\ will hold DBreeze for .NET35 (Windows only).

DBreeze version for .NET35 can be used only under Windows, cause utilizes system API FlushFileBuffers from kernel32.dll

DBreeze version for .NET40 doesn't use any system API functions and can be used under Linux MONO and under .NET 4>. For Windows, be sure to have latests .NET Framework starting from 4.5, because Microsoft has fixed bug with FileStream.Flush(true).

[20130608]

Restoring table from the other table.

Starting from DBreeze version 01.052 we can restore tables from the other source table on the fly.

The example code of compaction:

```
private void TestCompact()
    {

        using (var tran = engine.GetTransaction())
        {
            tran.Insert<int, int>("t1", 1, 1);
          tran.Commit();
        }


        DBreezeEngine engine2=new DBreezeEngine(@"D:\temp\DBreezeTest\DBR2")

        using (var tran = engine.GetTransaction())
        {
          tran.SynchronizeTables("t1");

          using (var tran2 = engine2.GetTransaction())
          {

        //Copying from main engine (Table t1) to engine2 (table "t1"), with changing all values to 2

            foreach (var row in tran.SelectForward<int,int>("t1"))
```

```
                {
                        tran2.Insert<int,int>("t1",row.Key,2);
                }

                    tran2.Commit();
            }

            engine2.Dispose();
//engine2 is fully closed.

            //moving table from engine2 (physical name) to main engine (logical name)

            tran.RestoreTableFromTheOtherFile("t1", @"D:\temp\DBreezeTest\DBR2\10000000");
            //Point555
        }

//Checking

        using (var tran = engine.GetTransaction())
        {
            foreach (var row in tran.SelectBackward<int,int>("t1"))
            {
//GETTING KEY 2
                Console.WriteLine("Key: {0}", row.Key);
            }
        }

    }
```

Up to point555 everything was ok, while copying data from one engine into another, parallel threads could read data from table "t1" of the main engine, parallel writing threads of course were blocked by tran.SynchronizeTables("t1"); command.

Starting from point555 some parallel threads which were reading table "t1" could have in memory reference to the old physical file, reading values from such references can bring to DBreeze TABLE_WAS_CHANGED_LINKS_ARE_NOT_ACTUAL exception.

Discussion link is [lock discussion copy](#)

Note: DON'T USE COMMIT AFTER RestoreTableFromTheOtherFile COMMAND, just close transaction.

[20130613]

Full tables locking inside of transactions.

Parallel threads can open transactions and in parallel read the same tables, in our standard configuration. For writing threads we use the tran.SynchronizeTables command to sequence writing threads access to the tables.

But what if we want to block access to the tables even in parallel reading threads, while modification commands of our current transaction are not yet finished?

For this we have developed a special type of transaction.

```
using (var tran = engine.GetTransaction(eTransactionTablesLockTypes.EXCLUSIVE, "t1", "p*", "c$"))
        {
                tran.Insert<int, string>("t1", 1, "Kesha is a good parrot");
        tran.Commit();
        }

using (var tran = engine.GetTransaction(eTransactionTablesLockTypes.SHARED, "t1"))
        {
                tran.Insert<int, string>("t1", 1, "Kesha is VERY a good parrot");
        tran.Commit();
        }
```

Inside of such a transaction we want to define the lock type for the listed tables.

Note, we must use either the first transaction type (engine.GetTransaction()) or new type (with SHARED/EXCLUSIVE) for the same tables among the whole program.

Example of usage:

```
        private void ExecF_003_1()
        {
        using (var tran = engine.GetTransaction(eTransactionTablesLockTypes.EXCLUSIVE, "t1",
"p*", "c$"))
        {
        Console.WriteLine("T1 {0}> {1}; {2}", DateTime.Now.Ticks,
System.Threading.Thread.CurrentThread.ManagedThreadId,
DateTime.Now.ToString("HH:mm:ss.ms"));
        tran.Insert<int, string>("t1", 1, "Kesha is a good parrot");
        tran.Commit();

        Thread.Sleep(2000);
        }
        }

        private void ExecF_003_2()
        {
```

```csharp
        List<string> tbls = new List<string>();
        tbls.Add("t1");
        tbls.Add("v2");
        using (var tran = engine.GetTransaction(eTransactionTablesLockTypes.SHARED,
tbls.ToArray()))
        {
        Console.WriteLine("T2 {0}> {1}; {2}", DateTime.Now.Ticks,
System.Threading.Thread.CurrentThread.ManagedThreadId,
DateTime.Now.ToString("HH:mm:ss.ms"));
        foreach (var r in tran.SelectForward<int, string>("t1"))
        {
                Console.WriteLine(r.Value);
        }
        }
        }


        private void ExecF_003_3()
        {
        using (var tran = engine.GetTransaction(eTransactionTablesLockTypes.SHARED, "t1"))
        {
        Console.WriteLine("T3 {0}> {1}; {2}", DateTime.Now.Ticks,
System.Threading.Thread.CurrentThread.ManagedThreadId,
DateTime.Now.ToString("HH:mm:ss.ms"));

        //This must be used in any case, when Shared threads can have parallel writes
        tran.SynchronizeTables("t1");

        tran.Insert<int, string>("t1", 1, "Kesha is a VERY good parrot");
        tran.Commit();

        foreach (var r in tran.SelectForward<int, string>("t1"))
        {
                Console.WriteLine(r.Value);
        }
        }
        }




using DBreeze.Utils.Async;


        private void testF_003()
        {

        Action t2 = () =>
        {
        ExecF_003_2();
        };
```

```
    t2.DoAsync();


    Action t1 = () =>
    {
    ExecF_003_1();
    };


    t1.DoAsync();


    Action t3 = () =>
    {
    ExecF_003_3();
    };


    t3.DoAsync();
    }
```

Transactions marked as SHARED will be executed in parallel. EXCLUSIVE transactions will wait till other transactions, consuming the same tables, are stopped and then block access for other threads (reading or writing) to the consuming tables.

This approach is good for avoiding transaction exceptions, in case of data compaction or removing keys with file re-creation, described in the previous chapter.

[20130811]

Remove KeyValue and get deleted value and notification if value exists in one round.

For this we have added overload in Master and in Nested tables: **RemoveKey**<TKey>(string tableName, TKey key, out bool WasRemoved, out byte[] deletedValue)

[20130812]

Insert key overload for Master and Nested table, letting not to overwrite key if it already exists.

For this we have added overload in Master and in Nested tables:

public void **Insert**<TKey, TValue>(string tableName, TKey key, TValue value, out byte[] refToInsertedValue, out bool **WasUpdated**, bool **dontUpdateIfExists**)

WasUpdated will become true, if value exists, and false if such value is not in DB. **dontUpdateIfExists,** equal to true, will not give DB to make an update.

Speeding up select operations and traversals with ValuesLazyLoadingIsOn.

DBreeze uses lazy value loading technique. For example, we can say
var row = transaction.Select<int,int>("t1",1);
At this moment we receive a row. We know that such row exists by row.Exists property and we know its key by row.Key property. At this moment value is still not taken into memory from disk. It will be read out from DB only when we instruct row.Value.

Sometimes it is good, when for us the only key is enough. Such cases can happen when we store a secondary index and the link to the primary table, as a part of the key. Or if we have "multiple columns" in one row. We need to get only one column and don't need to get complete, probably huge, value.

Nevertheless, lazy load will work a bit slower, in comparison with getting key and value in one round, due to extra HDD hits.

For this case we have developed in transaction a property/switch tran.**ValuesLazyLoadingIsOn**. By default it is ON (true), just set it to false and all transaction traversal commands, like SelectForwards, Backwards etc., will return us row already with a read out Value. This switch will also influence NestedTables which we get from tran.InsertTable, SelectTable and row.GetTable.
We can set this switch many times within one transaction to tune the speed of different queries.

[20140603]

Storing byte[] serialized objects as value, native support.

Starting from now we can bind any byte[] serializer/deserializer to DBreeze in following manner:

This declaration must be done right after DBreeze instantiation, before its real usage.

where...

We use mostly Protobuf.NET serializers in our projects. So an example will be done also with Protobuf. Get it via Nuget or make reference to it (protobuf-net.dll) .

Here are custom wrapping functions for Protobuf:

```csharp
    public static T DeserializeProtobuf<T>(this byte[] data)
    {
        T ret = default(T);

        using (System.IO.MemoryStream ms = new System.IO.MemoryStream(data))
        {

            ret = ProtoBuf.Serializer.Deserialize<T>(ms);
            ms.Close();
        }

        return ret;
    }


    public static object DeserializeProtobuf(byte[] data, Type T)
    {
        object ret = null;
        using (System.IO.MemoryStream ms = new System.IO.MemoryStream(data))
        {

            ret = ProtoBuf.Serializer.NonGeneric.Deserialize(T, ms);
            ms.Close();
        }

        return ret;
    }
    public static byte[] SerializeProtobuf(this object data)
    {
        byte[] bt = null;
        using (System.IO.MemoryStream ms = new System.IO.MemoryStream())
        {
            ProtoBuf.Serializer.NonGeneric.Serialize(ms, data);
            bt = ms.ToArray();
            ms.Close();
        }

        return bt;
    }
```

Now let's prepare an object for storing in DBreeze, decorated with Protobuf attributes (extra

documentation about protobuf can be found on its website):

```
[ProtoBuf.ProtoContract]
    public class XYZ
    {
      public XYZ()
      {
        P1 = 12;
        P2 = "sdfs";
      }


      [ProtoBuf.ProtoMember(1, IsRequired = true)]
      public int P1 { get; set; }


      [ProtoBuf.ProtoMember(2, IsRequired = true)]
      public string P2 { get; set; }
    }
```

And now let's use DBreeze for storing object:

```
using (var tran = engine.GetTransaction())
    {
        tran.Insert<int, XYZ>("t1", 1, new XYZ() {  P1 = 44, P2 = "well"});
        tran.Commit();
    }
```

And for retrieving object:

```
XYZ obj = null;

using (var tran = engine.GetTransaction())
    {
        var row = tran.Select<int, XYZ>("t1", 1);
        if (row.Exists)
        {
            obj = row.Value;
```

//!!! NOTE: better to assign row.Value to "obj" and then use "obj" among the program.
//Calling row.Value causes to rereading data from the table in case of default
//ValueLazyLoadingIsOn

```
            }
        }
```

[20160304]

Example of DBreeze initialization for UWP Universal Windows Platform.

```
string dbr_path =
System.IO.Path.Combine(Windows.Storage.ApplicationData.Current.LocalFolder.Path, "db");

        Task.Run(() =>
        {
        //System.Diagnostics.Debug.WriteLine(dbr_path );

        if (engine == null)
                engine = new DBreezeEngine(dbr_path );


        using (var tran = engine.GetTransaction())
        {
                tran.Insert<int, int>("t1", 1, 1);
                tran.Commit();
        }

        using (var tran = engine.GetTransaction())
        {
                var re = tran.Select<int, int>("t1", 1);
                System.Diagnostics.Debug.WriteLine(re.Value);
        }
        });
```

[20160320]

Quick start guide. Customers and orders

In this guide we will create customers, prototypes of business orders for these customers and determine different search functions.
Let's create a WinForm application, add NuGet reference to protobuf-net and DBreeze. On the form create a button and replace code of the form with this one:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
```

```csharp
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

using DBreeze;
using DBreeze.Utils;

namespace DBreezeQuickStart
{


    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        public static DBreeze.DBreezeEngine engine = null;

        protected override void OnFormClosing(FormClosingEventArgs e)
        {
            base.OnFormClosing(e);

            if (engine != null)
                engine.Dispose();
        }

        void InitDb()
        {
            if (engine == null)
            {
                engine = new DBreezeEngine(new DBreezeConfiguration { DBreezeDataFolderName =
@"S:\temp\DBreezeTest\DBR1" });
                //engine = new DBreezeEngine(new DBreezeConfiguration { DBreezeDataFolderName =
@"C:\temp" });

                //Setting default serializer for DBreeze
                DBreeze.Utils.CustomSerializator.ByteArraySerializator =
ProtobufSerializer.SerializeProtobuf;
                DBreeze.Utils.CustomSerializator.ByteArrayDeSerializator =
ProtobufSerializer.DeserializeProtobuf;
            }
        }


        [ProtoBuf.ProtoContract]
        public class Customer
```

```csharp
{
    [ProtoBuf.ProtoMember(1, IsRequired = true)]
    public long Id { get; set; }

    [ProtoBuf.ProtoMember(2, IsRequired = true)]
    public string Name { get; set; }
}

[ProtoBuf.ProtoContract]
public class Order
{
    public Order()
    {
        udtCreated = DateTime.UtcNow;
    }

    [ProtoBuf.ProtoMember(1, IsRequired = true)]
    public long Id { get; set; }

    [ProtoBuf.ProtoMember(2, IsRequired = true)]
    public long CustomerId { get; set; }

    /// <summary>
    /// Order datetime creation
    /// </summary>
    [ProtoBuf.ProtoMember(3, IsRequired = true)]
    public DateTime udtCreated { get; set; }
}

/// <summary>
/// -------------------------------------    STARTING TEST HERE   ------------------------------------
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button1_Click(object sender, EventArgs e)
{
    //One time db init
    this.InitDb();

    //Simple test

    ////Test insert
    //using (var tran = engine.GetTransaction())
    //{
    //    tran.Insert<int, int>("t1", 1, 1);
    //    tran.Insert<int, int>("t1", 1, 2);
    //    tran.Commit();
    //}

    ////Test select
    //using (var tran = engine.GetTransaction())
```

```csharp
//{
//    var xrow = tran.Select<int, int>("t1",1);
//    if (xrow.Exists)
//    {
//        Console.WriteLine(xrow.Key.ToString() + xrow.Value.ToString());
//    }

//    //or

//    foreach (var row in tran.SelectForward<int, int>("t1"))
//    {
//        Console.WriteLine(row.Value);
//    }
//}


//More complex test

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  RUN ONLY ONCE, THEN CLEAR DB

//Inserting CustomerId 1
var customer = new Customer() { Name = "Tino Zanner" };
Test_InsertCustomer(customer);

//Inserting orders for customer 1

//for (int i = 0; i < 5; i++)
//{
//    Test_InsertOrder(new Order { CustomerId = customer.Id });
//}
//or inserting batch of orders
Test_InsertOrders(
    Enumerable.Range(1, 5)
    .Select(r => new Order { CustomerId = customer.Id })
    );


//Inserting CustomerId 2
customer = new Customer() { Name = "Michael Hinze" };
Test_InsertCustomer(customer);

//Inserting orders for customer 2

//for (int i = 0; i < 8; i++)
//{
//    Test_InsertOrder(new Order { CustomerId = customer.Id });
//}
//or inserting batch of orders
Test_InsertOrders(
    Enumerable.Range(1, 8)
    .Select(r => new Order { CustomerId = customer.Id })
```

```csharp
    );

//Getting all orders
Console.WriteLine("All orders");
Test_GetOrdersByDateTime(DateTime.MinValue, DateTime.MaxValue);

//Getting Orders of customer 1
Console.WriteLine("Orders of customer 1");
Test_GetOrdersByCustomerIdAndDateTime(1, DateTime.MinValue, DateTime.MaxValue);

//Getting Orders of customer 2
Console.WriteLine("Orders of customer 2");
Test_GetOrdersByCustomerIdAndDateTime(2, DateTime.MinValue, DateTime.MaxValue);

/*
Result:
Inserted CustomerId: 1, Name: Tino Zanner
Inserted CustomerId: 2, Name: Michael Hinze
All orders
28.08.2015 07:15:57.734 orderId: 1
28.08.2015 07:15:57.740 orderId: 2
28.08.2015 07:15:57.743 orderId: 3
28.08.2015 07:15:57.743 orderId: 4
28.08.2015 07:15:57.743 orderId: 5
28.08.2015 07:15:57.757 orderId: 6
28.08.2015 07:15:57.758 orderId: 7
28.08.2015 07:15:57.758 orderId: 8
28.08.2015 07:15:57.759 orderId: 9
28.08.2015 07:15:57.759 orderId: 10
28.08.2015 07:15:57.759 orderId: 11
28.08.2015 07:15:57.760 orderId: 12
28.08.2015 07:15:57.760 orderId: 13
Orders of customer 1
28.08.2015 07:15:57.734 orderId: 1
28.08.2015 07:15:57.740 orderId: 2
28.08.2015 07:15:57.743 orderId: 3
28.08.2015 07:15:57.743 orderId: 4
28.08.2015 07:15:57.743 orderId: 5
Orders of customer 2
28.08.2015 07:15:57.757 orderId: 6
28.08.2015 07:15:57.758 orderId: 7
28.08.2015 07:15:57.758 orderId: 8
28.08.2015 07:15:57.759 orderId: 9
28.08.2015 07:15:57.759 orderId: 10
28.08.2015 07:15:57.759 orderId: 11
28.08.2015 07:15:57.760 orderId: 12
28.08.2015 07:15:57.760 orderId: 13
*/
return;
```

```csharp
        }




        /// <summary>
        /// 
        /// </summary>
        /// <param name="cust"></param>
        void Test_InsertCustomer(Customer cust)
        {
            try
            {
                using (var tran = engine.GetTransaction())
                {
                    //We don't need this line because we write only into one root table.
                    //Add more table names for safe transaction operations among multiple
                    //root tables (read docu)
                    tran.SynchronizeTables("Customers");


                    //In table Customers under key 1 we will have nested table with customers
                    var tbl = tran.InsertTable<int>("Customers", 1, 0);
                    //Under index 2 we will have monotonically grown id

                    if (cust.Id < 1)
                    {
                        //Insert
                        //Getting new ID for the customer
                        cust.Id = tran.Select<int, long>("Customers", 2).Value + 1;
                        //and inserting id back into key 2
                        tran.Insert<int, long>("Customers", 2, cust.Id);
                    }

                    //Inserting or updating of the customer
                    tbl.Insert<long, Customer>(cust.Id, cust);

                    //Committing entry
                    tran.Commit();
                }

                //Checking if customer is saved
                using (var tran = engine.GetTransaction())
                {
                    //using SelectTable instead of InsertTable (read docu). In short if we plan to write
and/or to read
                    //from nested table during one transaction then we use InsertTable, if only to read -
then SelectTable.
                    var tbl = tran.SelectTable<int>("Customers", 1, 0);
                    var row = tbl.Select<long, Customer>(cust.Id);
                    if (row.Exists)
```

```csharp
                Console.WriteLine("Inserted CustomerId: {0}, Name: {1}", row.Value.Id,
row.Value.Name);
            else
                Console.WriteLine("Insert failed");
        }
    }
    catch (Exception)
    {

        throw;
    }

}

/// <summary>
///
/// </summary>
/// <param name="order"></param>
void Test_InsertOrder(Order order)
{
    try
    {
        /*
        In our case, we will store orders of all customers in one table "Orders".
        Of course we could create for every customer his own table, like Order1, Order2...etc

        Later we are planning to search orders:
            1. by Order.Id
            2. by Order.udtCreated From-To
            3. by Order.CustomerId and Order.udtCreated From-To

            To fulfill 2 and 3 conditions we will need to store several extra indicies.

        */
        using (var tran = engine.GetTransaction())
        {
            //We don't need this line because we write only into one root table.
            //Add more table names for safe transaction operations among multiple
            //root tables (read docu)
            tran.SynchronizeTables("Orders");

            //Under key 1 we want to store nested table with orders
            var tbl = tran.InsertTable<int>("Orders", 1, 0);
            //Under key 2 we will store monotonically grown id for orders

            //Index table for the second search condition under key 3
            var tblDateIndex = tran.InsertTable<int>("Orders", 3, 0);
            //Index table for the third search condition under key 4
            var tblCustomerAndDateIndex = tran.InsertTable<int>("Orders", 4, 0);

            byte[] key = null;
```

```csharp
            if (order.Id < 1)
            {
                //Insert, getting new ID
                order.Id = tran.Select<int, long>("Orders", 2).Value + 1;
                //and inserting id back into index 2
                tran.Insert<int, long>("Orders", 2, order.Id);

                //Inserting secondary index into tblDateIndex.
                //Index will be complex and will let us search orders by creation DateTime
                key =
order.udtCreated.To_8_bytes_array().Concat(order.Id.To_8_bytes_array_BigEndian());
                //Here we have composite key date+uniqueOrderIndex (read docu). Value will be Id
of the order stored in tbl.
                //As a value we could also use the same order as in tbl (redundant storage for the
higher speed) or pointer to the key/value in tbl for SelectDirect (read docu)
                tblDateIndex.Insert<byte[], long>(key, order.Id);


                //Inserting secondary index into tblCustomerAndDateIndex
                //Key will start from CustomerId, then comes dateTime and then unique id of the
order
                key =
order.CustomerId.To_8_bytes_array_BigEndian().ConcatMany(order.udtCreated.To_8_bytes_array
(), order.Id.To_8_bytes_array_BigEndian());
                tblCustomerAndDateIndex.Insert<byte[], long>(key, order.Id);
            }

            //Inserting or updating customer
            tbl.Insert<long, Order>(order.Id, order);

            //Committing entry
            tran.Commit();
            }
        }
        catch (Exception)
        {

            throw;
        }
    }


    /// <summary>
    ///
    /// </summary>
    /// <param name="order"></param>
    void Test_InsertOrders(IEnumerable<Order> orders)
    {
        try
        {
```

```csharp
/*
In our case, we will store orders of all customers in one table "Orders".
Of course we could create for every customer his own table, like Order1, Order2...etc

Later we are planning to search orders:
    1. by Order.Id
    2. by Order.udtCreated From-To
    3. by Order.CustomerId and Order.udtCreated From-To

To fulfill 2 and 3 conditions we will need to store several extra indicies.

*/
using (var tran = engine.GetTransaction())
{

    //We don't need this line because we write only into one root table.
    //Add more table names for safe transaction operations among multiple
    //root tables (read docu)
    tran.SynchronizeTables("Orders");

    //Under key 1 we want to store nested table with orders
    var tbl = tran.InsertTable<int>("Orders", 1, 0);
    //Under key 2 we will store monotonically grown id for orders
    //Index table for the second search condition under key 3
    var tblDateIndex = tran.InsertTable<int>("Orders", 3, 0);
    //Index table for the third search condition under key 4
    var tblCustomerAndDateIndex = tran.InsertTable<int>("Orders", 4, 0);

    byte[] key = null;

    foreach (var ord in orders)
    {
        if (ord.Id < 1)
        {
            //Insert, getting new ID
            ord.Id = tran.Select<int, long>("Orders", 2).Value + 1;
            //and inserting id back into index 2
            tran.Insert<int, long>("Orders", 2, ord.Id);

            //Inserting secondary index into tblDateIndex.
            //Index will be complex and will let us search orders by creation DateTime
            key =
ord.udtCreated.To_8_bytes_array().Concat(ord.Id.To_8_bytes_array_BigEndian());
            //Here we have composite key date+uniqueOrderIndex (read docu). Value will be
Id of the order stored in tbl.
            //As a value we could also use the same order as in tbl (redundant storage for the
higher speed) or pointer to the key/value in tbl for SelectDirect (read docu)
            tblDateIndex.Insert<byte[], long>(key, ord.Id);

            //Inserting secondary index into tblCustomerAndDateIndex
            //Key will start from CustomerId, then comes dateTime and then unique id of the
```

```csharp
order
                key =
ord.CustomerId.To_8_bytes_array_BigEndian().ConcatMany(ord.udtCreated.To_8_bytes_array(),
ord.Id.To_8_bytes_array_BigEndian());
                tblCustomerAndDateIndex.Insert<byte[], long>(key, ord.Id);
            }

            //Inserting or updating customer
            tbl.Insert<long, Order>(ord.Id, ord);
        }

        //Committing all changes
        tran.Commit();
    }
}
catch (Exception)
{

    throw;
}
}

/// <summary>
///
/// </summary>
/// <param name="from"></param>
/// <param name="to"></param>
void Test_GetOrdersByDateTime(DateTime from, DateTime to)
{
    try
    {
        using (var tran = engine.GetTransaction())
        {
            var tbl = tran.SelectTable<int>("Orders", 1, 0);
            var tblDateIndex = tran.SelectTable<int>("Orders", 3, 0);

            byte[] keyFrom =
from.To_8_bytes_array().Concat(long.MinValue.To_8_bytes_array_BigEndian());
            byte[] keyTo =
to.To_8_bytes_array().Concat(long.MaxValue.To_8_bytes_array_BigEndian());

            foreach (var row in tblDateIndex.SelectForwardFromTo<byte[], long>(keyFrom, true,
keyTo, true))
            {
                var order = tbl.Select<long, Order>(row.Value);
                if (order.Exists)
                    Console.WriteLine(order.Value.udtCreated.ToString("dd.MM.yyyy HH:mm:ss.fff")
+ " orderId: " + order.Value.Id);
            }
        }
    }
```

```csharp
            catch (Exception)
            {

                throw;
            }
        }


    void Test_GetOrdersByCustomerIdAndDateTime(long customerId, DateTime from, DateTime
to)
        {
           try
           {
              using (var tran = engine.GetTransaction())
              {
                 var tbl = tran.SelectTable<int>("Orders", 1, 0);
                 var tblCustomerAndDateIndex = tran.SelectTable<int>("Orders", 4, 0);

                 byte[] keyFrom =
customerId.To_8_bytes_array_BigEndian().ConcatMany(from.To_8_bytes_array(),
long.MinValue.To_8_bytes_array_BigEndian());
                 byte[] keyTo =
customerId.To_8_bytes_array_BigEndian().ConcatMany(to.To_8_bytes_array(),
long.MaxValue.To_8_bytes_array_BigEndian());

                 foreach (var row in tblCustomerAndDateIndex.SelectForwardFromTo<byte[],
long>(keyFrom, true, keyTo, true))
                 {
                    var order = tbl.Select<long, Order>(row.Value);
                    if (order.Exists)
                       Console.WriteLine(order.Value.udtCreated.ToString("dd.MM.yyyy HH:mm:ss.fff")
+ " orderId: " + order.Value.Id);
                 }
              }
           }
           catch (Exception)
           {

               throw;
           }
        }

    }


    public static class ProtobufSerializer
    {
        /// <summary>
        /// Deserializes protobuf object from byte[]
        /// </summary>
        /// <typeparam name="T"></typeparam>
```

```csharp
/// <param name="data"></param>
/// <returns></returns>
public static T DeserializeProtobuf<T>(this byte[] data)
{
    T ret = default(T);

    using (System.IO.MemoryStream ms = new System.IO.MemoryStream(data))
    {

        ret = ProtoBuf.Serializer.Deserialize<T>(ms);
        ms.Close();
    }

    return ret;
}

/// <summary>
/// Deserializes protobuf object from byte[]. Non-generic style.
/// </summary>
/// <param name="data"></param>
/// <param name="T"></param>
/// <returns></returns>
public static object DeserializeProtobuf(byte[] data, Type T)
{
    object ret = null;
    using (System.IO.MemoryStream ms = new System.IO.MemoryStream(data))
    {

        ret = ProtoBuf.Serializer.NonGeneric.Deserialize(T, ms);
        ms.Close();
    }

    return ret;
}

/// <summary>
/// Serialize object using protobuf serializer
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
public static byte[] SerializeProtobuf(this object data)
{
    byte[] bt = null;
    using (System.IO.MemoryStream ms = new System.IO.MemoryStream())
    {
        ProtoBuf.Serializer.NonGeneric.Serialize(ms, data);
        bt = ms.ToArray();
        ms.Close();
    }

    return bt;
```

```
        }
    }
}
```

Quick start guide. Customers and orders. Object DBreeze

```csharp
using System;
using System.Collections.Generic;
using DBreeze;
using DBreeze.Utils;
using System.Linq;

namespace DBreezeExamples
{
    public class CustomersAndOrders
    {
        DBreezeEngine engine = null;

        public CustomersAndOrders(DBreezeEngine eng)
        {
            this.engine = eng;

            if(engine == null)
                engine = new DBreezeEngine(@"D:\Temp\x1");

            //Setting up NetJSON serializer (from NuGet) to be used by DBreeze
            DBreeze.Utils.CustomSerializator.ByteArraySerializator = (object o)
=> { return NetJSON.NetJSON.Serialize(o).To_UTF8Bytes(); };
            DBreeze.Utils.CustomSerializator.ByteArrayDeSerializator = (byte[]
bt, Type t) => { return NetJSON.NetJSON.Deserialize(t, bt.UTF8_GetString()); };
        }


        public class Customer
        {
            public long Id { get; set; } = 0;
            public string Name { get; set; }
        }


        public class Order
        {
            public Order()
            {
                udtCreated = DateTime.UtcNow;
            }
            public long Id { get; set; } = 0;
            public long CustomerId { get; set; }
            /// <summary>
```

```csharp
        /// Order datetime creation
        /// </summary>
        public DateTime udtCreated { get; set; }
    }


    public void Start()
    {
        //Inserting CustomerId 1
        var customer = new Customer() { Name = "Tino Zanner" };
        Test_InsertCustomer(customer);

        //Inserting some orders for this customer
        Test_InsertOrders(
            Enumerable.Range(1, 5)
            .Select(r => new Order { CustomerId = customer.Id })
            );

        //Test update order
        Test_UpdateOrder(3);

        //Inserting CustomerId 2
        customer = new Customer() { Name = "Michael Hinze" };
        Test_InsertCustomer(customer);

        //Inserting some orders for this customer
        Test_InsertOrders(
            Enumerable.Range(1, 8)
            .Select(r => new Order { CustomerId = customer.Id })
            );


        //------------------------- Various data retrieving

        //Getting Customer ById
        Test_GetCustomerById(2);

        //Getting Customer ByName
        Test_GetCustomerByFreeText("ichael");

        //Getting all orders
        Console.WriteLine("All orders");
        Test_GetOrdersByDateTimeRange(DateTime.MinValue, DateTime.MaxValue);

        //Getting Orders of customer 1
        Console.WriteLine("Orders of customer 1");
        Test_GetOrdersByCustomerIdAndDateTimeRange(1, DateTime.MinValue,
DateTime.MaxValue);

        ////Getting Orders of customer 2
```

```csharp
            Console.WriteLine("Orders of customer 2");
            Test_GetOrdersByCustomerIdAndDateTimeRange(2, DateTime.MinValue,
DateTime.MaxValue);

        }

        /// <summary>
        /// Inserting customer
        /// </summary>
        /// <param name="customer"></param>
        void Test_InsertCustomer(Customer customer)
        {
            try
            {
                /*
                    We are going to store all customers in one table
                    Later we are going to search customers by their IDs and
Names
                */

                using (var t = engine.GetTransaction())
                {
                    //Documentation https://goo.gl/Kwm9aq
                    //This line with a list of tables we need in case if we
modify more then 1 table inside of transaction
                    t.SynchronizeTables("Customers");

                    bool newEntity = customer.Id == 0;
                    if(newEntity)
                        customer.Id = t.ObjectGetNewIdentity<long>("Customers");

                    //Documentation https://goo.gl/YtWnAJ
                    t.ObjectInsert("Customers", new
DBreeze.Objects.DBreezeObject<Customer>
                    {
                        NewEntity = newEntity,
                        Entity = customer,
                        Indexes = new List<DBreeze.Objects.DBreezeIndex>
                        {
                            //to Get customer by ID
                            new DBreeze.Objects.DBreezeIndex(1,customer.Id) {
PrimaryIndex = true },
                        }
                    }, false);

                    //Documentation https://goo.gl/s8vtRG
                    //Setting text search index. We will store text-search
                    //indexes concerning customers in table "TS_Customers".
                    //Second parameter is a reference to the customer ID.
                    t.TextInsert("TS_Customers", customer.Id.ToBytes(),
```

```csharp
                customer.Name);

                //Committing entry
                t.Commit();
            }

        }
        catch (Exception ex)
        {
            throw ex;
        }

    }

    /// <summary>
    /// Inserting orders
    /// </summary>
    /// <param name="orders"></param>
    void Test_InsertOrders(IEnumerable<Order> orders)
    {
        try
        {
            /*
            We are going to store all orders from all customers in one
table.

            Later we are planning to search orders:
                1. by Order.Id
                2. by Order.udtCreated From-To
                3. by Order.CustomerId and Order.udtCreated From-To
            */

            using (var t = engine.GetTransaction())
            {
                //This line with a list of tables we need in case if we
modify morethen 1 table inside of transaction
                //Documentation https://goo.gl/Kwm9aq
                t.SynchronizeTables("Orders");

                foreach (var order in orders)
                {
                    bool newEntity = order.Id == 0;
                    if (newEntity)
                        order.Id = t.ObjectGetNewIdentity<long>("Orders");

                    t.ObjectInsert("Orders", new
DBreeze.Objects.DBreezeObject<Order>
                    {
                        NewEntity = newEntity,
                        Indexes = new List<DBreeze.Objects.DBreezeIndex>
                         {
```

```csharp
                                //to Get order by ID
                                new DBreeze.Objects.DBreezeIndex(1,order.Id) {
PrimaryIndex = true },

                                //to get orders in specified time interval
                                new
DBreeze.Objects.DBreezeIndex(2,order.udtCreated) { AddPrimaryToTheEnd = true },
//AddPrimaryToTheEnd by default is true
                                //to get orders in specified time range for
specific customer

                                new
DBreeze.Objects.DBreezeIndex(3,order.CustomerId, order.udtCreated)
                                },
                            Entity = order  //Setting entity
                        }, false);  //set last parameter to true, if batch
operation speed unsatisfactory
                    }

                    //Committing all changes
                    t.Commit();


                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }


        /// <summary>
        /// Updating order 3
        /// </summary>
        /// <param name="orderId"></param>
        void Test_UpdateOrder(long orderId)
        {
            try
            {

                using (var t = engine.GetTransaction())
                {
                    //This line with a list of tables we need in case if we
modify morethen 1 table inside of transaction
                    //Documentation https://goo.gl/Kwm9aq
                    t.SynchronizeTables("Orders");

                    var ord = t.Select<byte[], byte[]>("Orders",
1.ToIndex(orderId)).ObjectGet<Order>();
                    if (ord == null)
                        return;

                    ord.Entity.udtCreated = new DateTime(1977, 1, 1);
```

```csharp
                    ord.Indexes = new List<DBreeze.Objects.DBreezeIndex>()
                    {
                        //to Get order by ID
                            new
DBreeze.Objects.DBreezeIndex(1,ord.Entity.Id) { PrimaryIndex = true },
                            //to get orders in specified time interval
                            new
DBreeze.Objects.DBreezeIndex(2,ord.Entity.udtCreated), //AddPrimaryToTheEnd by
default is true
                            //to get orders in specified time range for
specific customer
                            new
DBreeze.Objects.DBreezeIndex(3,ord.Entity.CustomerId, ord.Entity.udtCreated)
                    };

                    t.ObjectInsert<Order>("Orders", ord, false);

                    //Committing all changes
                    t.Commit();

                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }


        void Test_GetOrdersByDateTimeRange(DateTime from, DateTime to)
        {
            Console.WriteLine("--------Test_GetOrdersByDateTimeRange--------");
            try
            {
                using (var t = engine.GetTransaction())
                {

                    //Documentation https://goo.gl/MbZAsB
                    foreach (var row in t.SelectForwardFromTo<byte[],
byte[]>("Orders",
                        2.ToIndex(from, long.MinValue), true,
                        2.ToIndex(to, long.MaxValue), true))
                    {
                        var obj = row.ObjectGet<Order>();
                        if (obj != null)
                            Console.WriteLine(obj.Entity.Id + " " +
obj.Entity.udtCreated.ToString("dd.MM.yyyy HH:mm:ss.fff") + " " +
obj.Entity.CustomerId);
                    }
```

```csharp
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
            Console.WriteLine("-------------------------------------");
        }


        void Test_GetOrdersByCustomerIdAndDateTimeRange(long customerId,
DateTime from, DateTime to)
        {

Console.WriteLine("--------Test_GetOrdersByCustomerIdAndDateTimeRange--------");
            try
            {
                using (var t = engine.GetTransaction())
                {


                    foreach (var row in t.SelectForwardFromTo<byte[],
byte[]>("Orders",
                        3.ToIndex(customerId, from, long.MinValue), true,
                        3.ToIndex(customerId, to, long.MaxValue), true))
                    {
                        var obj = row.ObjectGet<Order>();
                        if (obj != null)
                            Console.WriteLine(obj.Entity.Id + " " +
obj.Entity.udtCreated.ToString("dd.MM.yyyy HH:mm:ss.fff") + " " +
obj.Entity.CustomerId);
                    }

                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
            Console.WriteLine("-------------------------------------");
        }


        void Test_GetCustomerById(long customerId)
        {
            Console.WriteLine("--------Test_GetCustomerById--------");
            try
            {
                using (var t = engine.GetTransaction())
                {
```

```csharp
                    var obj = t.Select<byte[], byte[]>("Customers",
    1.ToIndex(customerId)).ObjectGet<Customer>();
                    if (obj != null)
                        Console.WriteLine(obj.Entity.Id + " " +
    obj.Entity.Name);
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
            Console.WriteLine("-----------------------------------");
        }

        /// <summary>
        /// Test_GetCustomerByFreeText
        /// </summary>
        /// <param name="text"></param>
        void Test_GetCustomerByFreeText(string text)
        {
            Console.WriteLine("--------Test_GetCustomerByFreeText--------");
            try
            {
                using (var t = engine.GetTransaction())
                {
                    foreach (var doc in
    t.TextSearch("TS_Customers").BlockAnd(text).GetDocumentIDs())
                    {
                        var obj = t.Select<byte[], byte[]>("Customers",
    1.ToIndex(doc)).ObjectGet<Customer>();
                        if (obj != null)
                            Console.WriteLine(obj.Entity.Id + " " +
    obj.Entity.Name);
                    }

                }
            }
            catch (Exception ex)
            {
                throw ex;
            }

            Console.WriteLine("-----------------------------------");
        }
    }
}
```

Output:

```
Test_GetCustomerById

2 Michael Hinze
```

Test_GetCustomerByFreeText

2 Michael Hinze

All orders

Test_GetOrdersByDateTimeRange

3 01.01.1977 00:00:00.000 1

1 24.03.2017 10:26:03.411 1

2 24.03.2017 10:26:03.517 1

4 24.03.2017 10:26:03.518 1

5 24.03.2017 10:26:03.518 1

6 24.03.2017 10:26:04.191 2

7 24.03.2017 10:26:04.191 2

8 24.03.2017 10:26:04.191 2

9 24.03.2017 10:26:04.191 2

10 24.03.2017 10:26:04.191 2

11 24.03.2017 10:26:04.191 2

12 24.03.2017 10:26:04.191 2

13 24.03.2017 10:26:04.191 2

Orders of customer 1

Test_GetOrdersByCustomerIdAndDateTimeRange

3 01.01.1977 00:00:00.000 1

1 24.03.2017 10:26:03.411 1

2 24.03.2017 10:26:03.517 1

4 24.03.2017 10:26:03.518 1

5 24.03.2017 10:26:03.518 1

Orders of customer 2

Test_GetOrdersByCustomerIdAndDateTimeRange

6 24.03.2017 10:26:04.191 2

7 24.03.2017 10:26:04.191 2

8 24.03.2017 10:26:04.191 2

9 24.03.2017 10:26:04.191 2

10 24.03.2017 10:26:04.191 2

11 24.03.2017 10:26:04.191 2

12 24.03.2017 10:26:04.191 2

13 24.03.2017 10:26:04.191 2

```csharp
using DBreeze;
using DBreeze.Utils;

DBreezeEngine engine = null;

 if (engine == null)
                engine = new DBreezeEngine(@"D:\Temp\x1");

//Setting up NetJSON serializer (from NuGet) to be used by DBreeze
DBreeze.Utils.CustomSerializator.ByteArraySerializator = (object o) => { return
NetJSON.NetJSON.Serialize(o).To_UTF8Bytes(); };
DBreeze.Utils.CustomSerializator.ByteArrayDeSerializator = (byte[] bt, Type t)
=> { return NetJSON.NetJSON.Deserialize(t, bt.UTF8_GetString()); };

public class Song
        {
            public long Id { get; set; }
            public string Path { get; set; } = "";
            public string ArtistName { get; set; } = "";
            public long ArtistId { get; set; }
            public DateTime SongReleaseDate { get; set; }
            public string Titel { get; set; }
            public string Album { get; set; }

            /// <summary>
            /// Helper, forming the view of the song to be searched via
text-search engine
            /// </summary>
            /// <returns></returns>
            public string ContainsText()
            {
                return ArtistName + " " + Titel + " "+ Album;
            }

        }

//inserting
            using (var t = engine.GetTransaction())
            {
                Song s = new Song
                {
                    Id = t.ObjectGetNewIdentity<long>("Songs"),
                    ArtistId = 1,
                    ArtistName = "The Beatles",
                    Album = "Revolver",
                    Path = @"C:\Songs\786788779.mp3",
                    SongReleaseDate = new DateTime(1966, 9, 5),
                    Titel = "Eleanor Rigby"
                };
```

```csharp
//Next command can be put as a function into DAL
            t.ObjectInsert<Song>("Songs", new
DBreeze.Objects.DBreezeObject<Song>
                {
                    NewEntity = true,
                    Entity = s,
//Using standard indexes for range queries
                    Indexes = new List<DBreeze.Objects.DBreezeIndex>
                        {
//unique song index, will be added to the end of all secondary indexes and gives
ability to pick an entity (song) by id
                            new DBreeze.Objects.DBreezeIndex(1,s.Id) {
PrimaryIndex = true },
//will give us ability to search any db song by release dates range
                            new
DBreeze.Objects.DBreezeIndex(2,s.SongReleaseDate),
//will give ability to search songs per Artist ordered by release date
                            new DBreeze.Objects.DBreezeIndex(3,s.ArtistId,
s.SongReleaseDate)
                        }
                });

            //Using text-search engine for the free text search
            t.TextInsert("SongsText", s.Id.To_8_bytes_array_BigEndian(),
s.ContainsText());

            s = new Song
            {
                Id = t.ObjectGetNewIdentity<long>("Songs"),
                ArtistId = 1,
                ArtistName = "The Beatles",
                Album = "Revolver",
                Path = @"C:\Songs\786788780.mp3",
                SongReleaseDate = new DateTime(1966, 9, 5),
                Titel = "Yellow Submarine"
            };

            t.ObjectInsert<Song>("Songs", new
DBreeze.Objects.DBreezeObject<Song>
            {
                NewEntity = true,
                Entity = s,
                Indexes = new List<DBreeze.Objects.DBreezeIndex>
                    {
                        new DBreeze.Objects.DBreezeIndex(1,s.Id) {
PrimaryIndex = true },
                        new
DBreeze.Objects.DBreezeIndex(2,s.SongReleaseDate),
                        new
DBreeze.Objects.DBreezeIndex(3,s.ArtistId,s.SongReleaseDate)
```

```
                }
            });

            t.TextInsert("SongsText", s.Id.To_8_bytes_array_BigEndian(),
s.ContainsText());

            s = new Song
            {
                Id = t.ObjectGetNewIdentity<long>("Songs"),
                ArtistId = 2,
                ArtistName = "Queen",
                Album = "Jazz",
                Path = @"C:\Songs\786788781.mp3",
                SongReleaseDate = new DateTime(1978, 11, 10),
                Titel = "Bicycle Race"
            };

            t.ObjectInsert<Song>("Songs", new
DBreeze.Objects.DBreezeObject<Song>
            {
                NewEntity = true,
                Entity = s,
                Indexes = new List<DBreeze.Objects.DBreezeIndex>
                    {
                        new DBreeze.Objects.DBreezeIndex(1,s.Id) {
PrimaryIndex = true },
                        new
DBreeze.Objects.DBreezeIndex(2,s.SongReleaseDate),
                        new
DBreeze.Objects.DBreezeIndex(3,s.ArtistId,s.SongReleaseDate)
                    }
            });

            t.TextInsert("SongsText", s.Id.To_8_bytes_array_BigEndian(),
s.ContainsText());

            s = new Song
            {
                Id = t.ObjectGetNewIdentity<long>("Songs"),
                ArtistId = 2,
                ArtistName = "Queen",
                Album = "The Miracle",
                Path = @"C:\Songs\786788782.mp3",
                SongReleaseDate = new DateTime(1989, 05, 22),
                Titel = "I want it all"
            };

            t.ObjectInsert<Song>("Songs", new
DBreeze.Objects.DBreezeObject<Song>
            {
```

```csharp
                    NewEntity = true,
                    Entity = s,
                    Indexes = new List<DBreeze.Objects.DBreezeIndex>
                        {
                            new DBreeze.Objects.DBreezeIndex(1,s.Id) {
PrimaryIndex = true },
                            new
DBreeze.Objects.DBreezeIndex(2,s.SongReleaseDate),
                            new
DBreeze.Objects.DBreezeIndex(3,s.ArtistId,s.SongReleaseDate)
                        }
                });

                t.TextInsert("SongsText", s.Id.To_8_bytes_array_BigEndian(),
s.ContainsText());

                t.Commit();
            }


            //Fetching data

            using (var t = engine.GetTransaction())
            {
                //Show all titles
                foreach (var el in t.SelectForwardFromTo<byte[],
byte[]>("Songs", 1.ToIndex(long.MinValue), true, 1.ToIndex(long.MaxValue),
true))
                {
                    //Console.WriteLine(el.ObjectGet<Song>().Entity.Titel);
                }
                /*
Eleanor Rigby
Yellow Submarine
Bicycle Race
I want it all
                */

                //Show titles, released up to year 1975
                foreach (var el in t.SelectForwardFromTo<byte[],
byte[]>("Songs", 2.ToIndex(DateTime.MinValue), true, 2.ToIndex(new
DateTime(1975,1,1)), true))
                {
                    //Console.WriteLine(el.ObjectGet<Song>().Entity.Titel);
                }
                /*
                 Eleanor Rigby
                Yellow Submarine
                        */
```

```csharp
                //Show all Queen titles (Queen id is 2) in time range
                foreach (var el in t.SelectForwardFromTo<byte[],
byte[]>("Songs", 3.ToIndex((long)2,DateTime.MinValue), true,
3.ToIndex((long)2,DateTime.MaxValue), true))
                {
                    //Console.WriteLine(el.ObjectGet<Song>().Entity.Titel);
                }
                /*
                  Bicycle Race
                  I want it all
                 */

                //Show all Queen titles up to year 1983 (Queen id is 2)
                foreach (var el in t.SelectForwardFromTo<byte[],
byte[]>("Songs", 3.ToIndex((long)2, DateTime.MinValue), true, 3.ToIndex((long)2,
new DateTime(1983, 1, 1)), true))
                {
                    Console.WriteLine(el.ObjectGet<Song>().Entity.Titel);
                }
                /*
                  Bicycle Race
                 */


                //------------  search using text-search engine

                foreach (var el in
t.TextSearch("SongsText").Block("jazz").GetDocumentIDs())
                {
                    var o = t.Select<byte[], byte[]>("Songs",
1.ToIndex(el)).ObjectGet<Song>();
                        if (o != null)
                            Console.WriteLine(o.Entity.Titel);
                }
                /*
                 Bicycle Race
                 */

                foreach (var el in t.TextSearch("SongsText").Block("queen
rac").GetDocumentIDs())
                {
                    var o = t.Select<byte[], byte[]>("Songs",
1.ToIndex(el)).ObjectGet<Song>();
                        if (o != null)
                            Console.WriteLine(o.Entity.Titel);
                }
                /*
                 I want it all - comes inside because belongs to album The
Miracle
                 Bicycle Race
```

```
                          */


                }
```

[20160329]

DBreeze.DataStructures.DataAsTree

Due to the desire of some people to implement into DBreeze an ability to store data as a
tree, with dependent nodes, out of the box, we have created a new namespace
DBreeze.DataStructures. And inside there is a class DataAsTree.

How to work with that:

```csharp
using DBreeze;
using DBreeze.DataStructures;

DataAsTree rootNode = null;
            DataAsTree insertedNode = null;


            using (var tran = engine.GetTransaction())
            {
            //In this "testtree" we will store our new DataStructure, so it
should be synchronized with other tables,
            //if we want to modify it
            tran.SynchronizeTables("testtree");

            //Initializing root node. Must be initialized after any new
transaction (if DataAsTree must be used there)
            rootNode = new DataAsTree("testtree", tran);

            //Adding to the root node a single child node
            rootNode.AddNode(new DataAsTree("folder1"));
            //Inserting second child node, getting reference to inserted node
            insertedNode = rootNode.AddNode(new DataAsTree("folder2"));

            //Preparing a node batch
            var nodes = new List<DataAsTree>();
            nodes.Add(new DataAsTree("xfolder1"));
            nodes.Add(new DataAsTree("xfolder2"));
            //nodes.Add(new DataAsTree("xfolder2"));
```

```csharp
            nodes.Add(new DataAsTree("xfile1"));


            //And inserting it under the second root child node
            insertedNode.AddNodes(nodes);

            //Inserting node with the content (it can be counted as file, thou
any node can have Content)
            var fileNode = new DataAsTree("file1");
            fileNode.NodeContent = new byte[] { 1, 2, 3, 4, 5 };
            //Adding it also to the root
            rootNode.AddNode(fileNode);

            //Committing transaction, so all our changes are saved now
            tran.Commit();
            }//eo using
```

Ok, now let's iterate through nodes

```csharp
  using (var tran = engine.GetTransaction())
            {
            //Again creating rootnode (always when we start new transaction it
must be performed)
            rootNode = new DataAsTree("testtree",tran);

            //And recursively read all our inserted nodes starting from Root
(any node can be used)
            foreach (var tn in
rootNode.ReadOutAllChildrenNodesFromCurrentRecursively(tran))
            {
                Console.WriteLine(tn.NodeName + "_" + tn.NodeId + "_" +
tn.ParentNodeId);

                byte[] cnt = tn.GetContent(tran);
                if (cnt != null)
                {
                    //Showing content of the file

                }
            }
            }//eo using
```

Now, let's grab nodes by specified name and rebind them to other parent change them:

```csharp
using (var tran = engine.GetTransaction())
        {
            tran.SynchronizeTables("testtree");

            rootNode = new DataAsTree("testtree", tran);

            //Reconnecting all nodes from 2 parentId to 1 parentId
            foreach (var tn in rootNode.GetNodesByName("xf"))   //or
rootNode.GetFirstLevelChildrenNodesByParentId(2)
            {
                rootNode.RemoveNode(tn);

                tn.ParentNodeId = 1;
                rootNode.AddNode(tn);
            }

            tran.Commit();
        }//eo using
```

Now, let's rename nodes and supply different content

```csharp
using (var tran = engine.GetTransaction())
        {
            tran.SynchronizeTables("testtree");

            rootNode = new DataAsTree("testtree", tran);
```

```
                    //Renaming nodes and setting new content
                    foreach (var tn in rootNode.GetNodesByName("xf"))
                    {
                        tn.NodeName = tn.NodeName + "_new_";
                        tn.NodeContent = new byte[] { 7, 7, 7 };
                        rootNode.AddNode(tn);
                    }

                    tran.Commit();
                }//eo using
```

[20160602]

DBreeze and external synchronizers, like ReaderWriterLockSlim

In **different concurrent functions** of the application several approaches may be mixed e.g:

F1(){
   RWLS.ENTER_WRITE_LOCK

     DBREEZE.TRAN.START
      DBREEZE.SYNCTABLE("X")
        DO
     DBREEZE.TRAN.END

   RWLS.EXIT_WRITE_LOCK
}
F2(){
 DBREEZE.TRAN.START
   DBREEZE.SYNCTABLE("X")
      RWLS.ENTER_WRITE_LOCK

```
        DO
    RWLS.EXIT_WRITE_LOCK
    //OR
     RWLS.ENTER_READ_LOCK
        DO
    RWLS.EXIT_READ_LOCK

  DBREEZE.TRAN.END
}
```

There is a possibility of a **deadlock** in such parallel sequence:
F1.RWLS.ENTER_WRITE_LOCK
F2.DBREEZE.SYNCTABLE("X")
F1.DBREEZE.SYNCTABLE("X") - WAIT
F2. RWLS.ENTER_READ_LOCK - WAIT
DEADLOCK.

First simple rule to avoid is **not to mix approaches** in functions.
Having the fact that Dictionary is, a priori, must be faster than any persistent object and
access to it has to be designed as a super fast and concurrent,
there can be formulated a **RULE to use as shorter RWLS** (like in F2) **as possible**.
So, better, when RWLS always resides after SYNCTABLE

Also ConcurrentDictionary with AddOrUpdate may be considered.

[20160628]
Integrated document text search functionality out of the box into DBreeze core.

Starting from version 75, DBreeze has implemented a text search engine from the
DBreezeBased project. Let's assume, that we have following class:

```
class MyTask
    {
        public long Id { get; set; }
        public string Description { get; set; } = "";
        public string Notes { get; set; } = "";
    }
```

We want to store it in DBreeze, but also we want to be able to find it by the text, represented

in Description and Notes.

```csharp
            using (var tran = engine.GetTransaction())
             {

MyTask tsk = null;

                //we want to store searchable text (text index) in table
"TasksTextSearch" and MyTask itself in table "Tasks"
                tran.SynchronizeTables("Tasks", "TasksTextSearch");

                //Storing task
                tsk = new MyTask()
                {
                    Id = 1,
                    Description = "Starting with the .NET Framework version 2.0,
well if you derive a class from Random and override the Sample method, the
distribution provided by the derived class implementation of the Sample method
is not used in calls to the base class implementation of the NextBytes method.
Instead, the uniform",
                    Notes = "distribution returned by the base Random class is
used. This behavior improves the overall performance of the Random class. To
modify this behavior to call the Sample method in the derived class, you must
also override the NextBytes method"
                };
                tran.Insert<long, byte[]>("Tasks", tsk.Id, null);

                //Creating text, for the document search. any word or word part
(minimum 3 chars, check TextSearchStorageOptions) from Description and Notes
will return us this document in the future
                tran.TextInsert("TasksTextSearch",
tsk.Id.To_8_bytes_array_BigEndian(), tsk.Description + " " + tsk.Notes, "");


                tsk = new MyTask()
                {
                    Id = 2,
                    Description = "VI guess in Universal Apps for Xamarin you
need to include the assembly when loading embedded resources. I had to change",
                    Notes = "I work on.NET for UWP.This is super interesting and
I'd love to take a deeper look at it after the holiday. If "
                };
                tran.Insert<long, byte[]>("Tasks", tsk.Id, null);
                tran.TextInsert("TasksTextSearch",
tsk.Id.To_8_bytes_array_BigEndian(), tsk.Description + " " + tsk.Notes, "");


                tsk = new MyTask()
```

```
                {
                    Id = 3,
                    Description = "Redistribution and use in source and binary
 forms, with or without modification, are permitted provided that the following
 conditions are met",
                    Notes = "This clause was objected to on the grounds that as
 people well changed the license to reflect their name or organization it led to
 escalating advertising requirements when programs were combined together in a
 software distribution: every occurrence of the license with a different name
 required a separate acknowledgment. In arguing against it, Richard Stallman has
 stated that he counted 75 such acknowledgments "
                };
                tran.Insert<long, byte[]>("Tasks", tsk.Id, null);
                tran.TextInsert("TasksTextSearch",
 tsk.Id.To_8_bytes_array_BigEndian(), tsk.Description + " " + tsk.Notes, "");

                //Committing all together.
                  tran.Commit();
            }
```

Command "tran.TextInsert" acts like Insert/Update. **System** will **automatically remove** disappeared **words** and **add new** words from the supplied searchables set - **SMART UPDATE** OF **CHANGED WORDS ONLY**.

There are also extra commands:

"tran.TextAppend" - will append extra words to existing searchables-set

"tran.TextRemove" - will remove supplied words from existing searchables-set (full-match words only)

and

"tran.TextRemoveAll" - will completely remove document from searchables

Command "tran.TextInsert accepts as second parameter external documentID (as byte[]) which will be returned as a search result when we are searching results via tran.TextSearch..block..GetDocumentIDs().

"contains" or "full-match"

Very important term that we have discovered is the way how we store searching text.

tran. TextInsert(string tableName, byte[] documentId, string containsWords, string fullMatchWords = "", bool deferredIndexing = false, int containsMinimalLength = 3)

There is a possibility to store words which can be later searched by using **"contains" logic** and by **"full-match" logic.** Words stored by "full-match" reside in less area in the database file (memory) and can be searched only by searching complete words. This is necessary for multi-parameter search, which will be explained in later chapters.

Example:

tran. TextInsert(string tableName, byte[] documentId, "wizard", "table", false, 3)

The word "table" will be written only once.

The word "wizard" will be presented as
"wizard",
"izard",
"zard",
"ard" (because containsMinimalLength parameter = 3)

Search engine, internally using StartsWith, will be able to find a match by words
wizard, wizar, wiza, wiz, izard, izar, iza, zar, zard, ard etc…

Deferred indexing

By default every insert into text will be with option DefferedIndexing = false

```
 tran.TextInsert("TaskFullTextSearch", tsk.Id.To_8_bytes_array_BigEndian(), tsk.Description + " " +
tsk.Notes);
```

It means that a search service is created within a given transaction, while committing it.

It's good for a relatively small amount of search words, but the larger this amount is, the longer it will take to commit a transaction.
To stay with the fast commits, independent of the searchable-set size, use **DefferedIndexing = true** option. It will **run** indexing **in parallel thread**.

In case of **abnormal program termination**, **indexing will go on after restarting** DBreeze engine.

It's possible to mix approaches for different searchable sets inside of one transaction, by changing **DefferedIndexing** parameter for different tran.TextInsertToDocument.

Storage configuration.

TextSearch subsystem can be configured via engine configuration:
 DBreezeConfiguration config = new DBreezeConfiguration()
        {
           DBreezeDataFolderName = ...
           TextSearchConfig =new DBreezeConfiguration.TextSearchConfiguration
           {
              ...
           }
        };

Current quantity of words in one block is configured to 1000 and the initial reserved space for every block is 100.000 bytes.

Having that
Minimal size of the block is 100.000 bytes.
The Maximum size of the block for 10.000 added documents is 1.250.000 bytes.
Expected size of the of the block for 10.000 added documents is 300,000 bytes

For mobile development it is recommended to decrease some values:
E.g.
TextSearchConfig =new DBreezeConfiguration.TextSearchConfiguration
        {
          QuantityOfWordsInBlock = 100,
          MinimalBlockReservInBytes = 1000

        }

Read how to search documents here

[20160718]
.NET Portable support

Get from release folder Portable version of DBreeze (or correspondent version from GitHub Release):

https://github.com/hhblaze/DBreeze/releases/download/v1.075/DBreeze_01_075_20160705_NETPortable.zip

Now we are able to describe any business logic, relying on DBreeze manipulation, right in the portable (cross-platform) class and then to use the final library from any platform specific project (UWP, Android, iOS etc.).

.NET Portable doesn't have file operations implemented, that's why FSFactory.cs class (from

NETPortable.zip folder) must be instantiated in a platform specific class and then, like an implementing interface parameter, supplied to a portable DBreeze instance. Read more in !!!Readme.txt (from NETPortable.zip folder).

[20160921]
DBreezeEngine.BackgroundTasksExternalNotifier

At this moment we have one possible background task - TextIndexer. Probably in the future we can have more of them.

To receive notifications about background tasks execution state it is enough to instantiate any object of type Action<string,object> and set it to BackgroundTasksExternalNotifier property of instantiated DBreezeEngine.

```
DBEngine.BackgroundTasksExternalNotifier = (name, obj) => {
        Console.WriteLine(name);
    };
```

First param will be actionName second any object.
Concerning **TextDeferredIndexer**. Generated actions with nullable second param are:
"**TextDefferedIndexingHasStarted**", "**TextDefferedIndexingHasFinished**"

[20161122]
New DBreeze Text-Search-Engine Insert/Search API. Explaining word aligned bitmap index and multi-parameter search via LogicalBlocks. "contains" and "full-match" logic.

Intro

New insert API is described here, please review it.

Starting from DBreeze version 1.080.2016.1122, we have enhanced possibilities of the integrated text-search engine.

It's possible to store words, which may be later searched by "contains" logic and by "full-match" logic all together.

Also it's possible to mix AND/OR/XOR/EXCLUDE logics while text search.
E.g. we can implement following search logic (& means AND, | means OR):

"(boy | girl) & (with) & (red | black) & (umbrella | shoes)"

All such sentences will match our search:
Boy with red umbrella
Boy with black umbrella
Boy with red shoes
Girl with red umbrella
Girl with red umbrella
Boy with red shoes and red umbrella
Etc…

Though sentences like
"Boys with reddish shoesssss"
"Girls without black umbrella"
May also be returned until we set up words to be searched only by full-match.

Multi-parameter complex index

Such an approach can help not only in search of the text, but also in the fast object search by multiple parameters, avoiding full scans or building unnecessary relational indexes.

We can imagine objects with many properties from different comboboxes, checkboxes and other fields which we want to search at once:

"(#GENDER_MAN) & (#CITY_HAMBURG) & (#LANG_RU | #LANG_EN) & (#LANG_DE) & (#PROF_1 | #PROF_2)"

Etc.. such search criteria list can grow.

#PROF_1 means profession or skill with database index 1 stored in a separate table (let it be programmer).
#PROF_2 means profession or skill with database index 2 (let it be unix administrator).

Here we are searching for a candidate, who must be a **man** from **Hamburg** with the knowledge of **German** language, extra both or any of **Russian** or **English languages** must be on board, who is a **programmer** or a **unix administrator**.

Range queries

To store data for the range traversing we must use ordinary DBreeze indexes, but with a

text-search subsystem we can make lots of tricks.

For example, we want to find Honda car dealers in a 300 km radius around Bremen.

Let's assume that we have one in Hamburg - 200km away from Bremen. To save its location to be searched via text-system, we split earth maps on tiles with the area of 500 km^2, receiving a grid with numbered tiles (like T12545). It's just a mathematical operation, by supplying the latitude and longitude of a point we can momentally get the tile name where it resides.

Before car dealer is stored into database, its address must be geocoded and tile number must be stored inside the text-search index together with the other meta information:

Insert "#PROD_HONDA #PROD_MITSUISHI #CITY_hamburg #TILE_T15578 "

So, this car dealer sells Honda and Mitsubishi, residing somewhere in tile T15578.

By searching any Honda dealer in radius 300 km from Bremen, geocoding Bremen city center coordinates and getting all tiles in radius 300km around this point (very fast operation, getting all square names from top-left corner to bottom-right). Let's assume that around this Bremen point, in radius 300 km, there are four 500 km^2 tiles (T14578 T14579 T15578 T15579).

Now we search
"(#PROD_HONDA) & (#TILE_T14578 | #TILE_T14579 | #TILE_T15578 | #TILE_T15579)"

A Hamburg car dealer will be found. Distance for returned entities may be re-checked to get 100% precision.

Note, it's possible to use "deep zoom" approach and store
together with 500 km^2 tile-system, location of a car dealer in 1000 km^2 and in 100 km^2 tile-systems:

Insert "#PROD_HONDA #PROD_MITSUISHI #CITY_hamburg #TILE_T15578 #TILE_G14578 #TILE_V45654"

Searching tiles will depend upon the radius.

*The same trick is possible with **DateTimes**.*

We can save in the text-index global DateTime information, like year and month, to make several types of combined search easier:

Insert "#CUSTOMER_124 #DT_YEAR_2016 #DT_MONTH_6 repairing monoblocks drinking

coffee and watching TV"

Finding documents for the customerID-124, from 2016 may - 2016 july, with the existent text "monoblocks":

"(#CUSTOMER_124) & (#DT_YEAR_2016) & (#DT_MONTH_5 | #DT_MONTH_6 | #DT_MONTH_7) & (monoblock)"

Of course, everything depends upon the quantity of data residing under different indexes. Sometimes it is better to traverse the range of CustomerID+DateTime DBreeze index, getting all documents and checking that they contain the word "monoblock" inside.

Parameter changes

In case if parameter changes it's enough to make new insert:

Was inserted "#GENDER_Man #CITY_BREMEN #STATE_SINGLE"
New insert "#GENDER_Man #CITY_BREMEN #STATE_MARRIED"

System will automatically remove #STATE_SINGLE and add #STATE_MARRIED connection to the document ID.
Unaltered words will not be touched ("smart" update).

Inserting/Updating with "contains" and "full-match"

To implement above ideas via DBreeze we have following tools:

```
using (var tran = xeng.GetTransaction())
            {

            tran.TextInsert("TextSearch", ((long)157).To_8_bytes_array_BigEndian(),
"Alex Solovyov Hamburg 21029", "#JOB_1 #STATE_3");

            tran.TextInsert("TextSearch", ((long)182).To_8_bytes_array_BigEndian(),
"Ivars Sudmalis Hamburg 21035", "#JOB_2 #STATE_2");

            tran.Commit();
            }
```

We have inserted 2 documents with external IDs 157 and 182.

Note, a new insert of the same external ID will work like an update.

Words " "Ivars Hamburg 21035" " "Alex Hamburg 21029" are stored using "contains" logic and later can be searched using "contains" logic. So, both documents can be found by searching text "mburg".

Words "#JOB_1 #JOB_2 #STATE_2 #STATE_3" are stored using "full-match" logic and can be found only by searching complete words.
E.g search by "ATE_" will not return these documents.

Programmers must take care that "contains" words are not being mixed with "full-match" words to avoid "dirty" search results.
E.g. it's better to disallow "contains" words like "whatever#JOB_1", otherwise it will be mixed with full-matched "#JOB_1".

Search by Logical Blocks


```
using (var tran = xeng.GetTransaction())
            {

//We have to instantiate search manager for the table first
            var tsm = tran.TextSearch("TextSearch");



            foreach (var w in
                        tsm.BlockAnd("mali 035", "")
            .And(tsm.BlockOr("many less", ""))
                        .Or(tsm.BlockAnd("", "21029"))
 .Exclude("","test")
                        .GetDocumentIDs())
            {
                Console.WriteLine(w.To_Int64_BigEndian());
            }
}
```


First, the search manager for the table (TSM) must be instantiated. It lives only inside of one transaction. Via TSM it's possible to receive logical blocks. Logical block is a set of space separated words which must be searched. Minimal quantity is 1 word. First parameter is "contains" words, second - "full-match" words. They can be mixed.

In our example:

(((mali & 035) & (many | less)) | (full-matched-word 21029)) then exclude all documents where exists full-matched *word "test"*

Inside of the block can be either AND or OR logic is used.
That's why TSM can return either BlockAnd or BlockOr.

Between blocks it is possible to make **AND, OR, XOR, EXCLUDE** operations.

To achieve
*"(boy | girl) & **with** & (red | black) & (**umbrella** | **shoes**)",*
where bolds - are full-matched, we could write:

```
 foreach (var w in
                        tsm.BlockOr("boy girl", "")
                        .And(tsm.BlockAnd("", "with"))
                        .And(tsm.BlockOr("red black", " "))
                        .And(tsm.BlockOr("", "umbrella shoes"))
                        .GetDocumentIDs())
            {
                Console.WriteLine(w.To_Int64_BigEndian());
            }
```

Some more possibilities of usage:

Blocks can be reused in case if we need to make several independent checks with the same set of search parameters:

```
 var tsm = tran.TextSearch("MyTextSearchTable");

var bl1 = tsm.BlockOr("boy girl", "");
var bl2 = tsm.Block("boy girl");
var bl3 = tsm.Block("","boy girl",false).And(tsm.Block("","left right"), false);


 foreach (var w in
 tsm.BlockAnd("2103")

 //FIrst block must be added via tsm,  born by tsm
//then blocks can be added in different formats

 .Or(new DBreeze.TextSearch.BlockAnd("2102"))
                        .And("","#LNDUA")
 .And(new DBreeze.TextSearch.BlockAnd("","#LNDUA"))
 .And(tsm.Block("boy girl","pet",false))
                        .And("","#LNDUA #LNDDE",false)
 .Exclude(bl2)

                        .GetDocumentIDs())

            {Console.WriteLine(w.To_Int64_BigEndian());}
```

Note, inserting/searching words are **case-insensitive.**

TextGetDocumentsSearchables

tran.TextGetDocumentsSearchables can help to understand which searchables are bound to concrete documents by supplied external IDs

How it works and overall performance

- If word is **stored** to be used with "**contains**" logic it will be saved like this:

    E.g. word "around" will be saved like

    around
    round
    ound
    und (up to minimal search length)

    Search by "**contains**" logic works like **DBreeze**.**StartsWith**, so "roun" - will find a piece of the word "around"
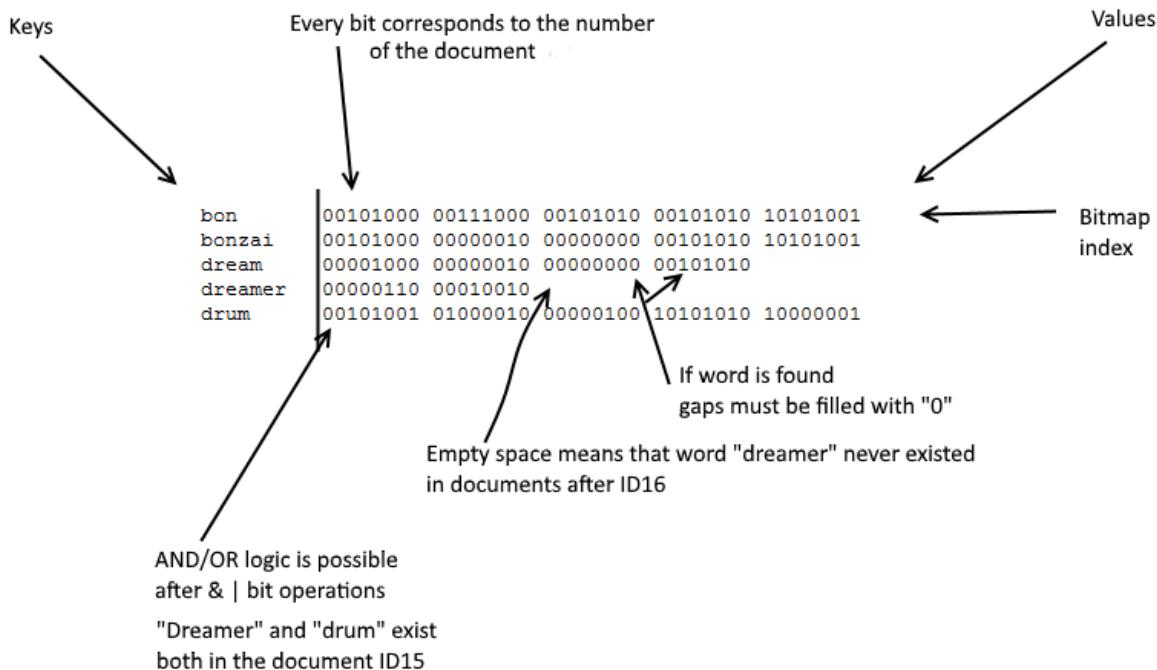
    If word is **stored** to using "**full-match**" logic it will be saved only once

    E.g. word "around" will be saved only once

    around

    Search by "**full-match**" logic uses **DBreeze.Select**. Such word will be found only by searching "around"

- External document IDs, supplied while inserting, will be transformed into monotonically grown internal document ID. Matching between them will be saved, inserted text will be also saved (it's necessary for the "smart" update).

- Word aligned bitmap index (DBreeze WABI).

```
bon      00101000 00111000 00101010 00101010 10101001          Bitmap
bonzai   00101000 00000010 00000000 00101010 10101001          index
dream    00001000 00000010 00000000 00101010
dreamer  00000110 00010010
drum     00101001 01000010 00000100 10101010 10000001
```

If word is found
gaps must be filled with "0"

Empty space means that word "dreamer" never existed
in documents after ID16

AND/OR logic is possible
after & | bit operations

"Dreamer" and "drum" exist
both in the document ID15

- When the word is stored into DBreeze table as a key, as a value we store byte[], where each bit location corresponds to the internal document ID and bit value 1 means that this word is inside of this document. If word exists in documents with internal IDs 1,2,3, 5,6,7 - WABI makes transformation into binary 11101110 (0xEE). It means, if there are 1 000 000 (one million) documents and there is a word that was found only in the latest document, we need 1000000/8 = 125000 bytes (125KB for its bitmap index). The same size for the bitmap index we need in case, if word exists in all documents. If word was found only in the first document it will reside only 1 byte. If there are 1mln documents and each of them has the same 1mln words in it, the final space must be around 125GB. But words are stored in blocks, WABIs are optimized and compressed, so the real physical space will be much less. If there are 20000 unique words which are dispersed across 10000 documents, then there must be around 7MB of space used, before optimization algorithms start to work.

- Search performance. *"(boy | girl) & **with** & (**red** | **black**) & (**umbrella** | **shoes**)"* , where bolds are full-matches, - for them, one DBreeze Select per word has to be made to get WABI before starting comparative analysis. For non-bold - SelectForwardStartsWith is used, the cursor may find more than one matching result. But, for easy computation, - as many search words, as many internal selects have to be made. Thereafter, received binary indexes have to be binary merged by binary AND/OR logic.

   In our example, 5 Selects and 2 SelectForwardStartsWith have to be made. StartsWith is always limited by **TextSearch..NoisyQuantity** parameter (default 1000). If any search word gets .NoisyQuantity limitation, **TextSearch...SearchCriterialsNoisy** will be set to true. It's just a flag that recommends precise search criteria.

Clustering

Insert into the text-search table is accompanied by supplying the index table name. Making a new index table, let's say, for every 50000 documents, will give the possibility to run search queries in parallel for every 50000 documents block.
Received results have to be merged.

[20161214]
Mixing of multi-parameter and a range search.

For example there are tasks with descriptions. Each task has a creation date in UTC.
We would like to search all those tasks, which creation dates stay in a specified time range and their descriptions contain some defined words.

We are not very powerful in limiting ranges, using our text-search system only, and in the key-range-search system we are not very powerful in finding multi-parameters, without the full-scan of the range.

Every new insert of the external-ID into the text-search subsystem generates an internal monotonically grown ID. So, in case we are sure that our external-IDs also grow up (maybe not monotonically, but grow up) with every insert, we can build up a mixed-search system.

Starting from version 1.81, it's possible to make by supplying optional external-IDs to the TextSearchTable object, limiting the search range. Also it's possible to choose the ordering of the returned document IDs (ascending, descending).
Default choice is always **descending** - latest inserted documents will be returned first in the text-search system.

Getting all document-IDs which contain words "boy" and "shoes" and are limited by external IDs from 3 - 17.

```
  var tsm = tran.TextSearch("MyTextSearchTable");
//new 3 optional parameter
            tsm.ExternalDocumentIdStart =
((long)3).To_8_bytes_array_BigEndian();
            tsm.ExternalDocumentIdStop =
((long)17).To_8_bytes_array_BigEndian();
            tsm.Descending = false;
```

```
        foreach (var w in
        tsm.BlockAnd("boy shoes")
        .GetDocumentIDs())
        {
                Console.WriteLine(w.To_Int64_BigEndian());
        }
```

Note, when Desending=true, Start and Stop change their places:

//Thinking descending
        tsm.ExternalDocumentId**Start** = ((long)**17**).To_8_bytes_array_BigEndian();
        tsm.ExternalDocumentId**Stop** = ((long)**3**).To_8_bytes_array_BigEndian();
        tsm.Descending = true;

In our example, we could create an entity "task", then create a secondary index, building a combined index from "creation DateTime"+"task Id", then insert description into "TaskSearchTable", supplying as external-ID the "task ID".

When time to search by description and a time range comes, we could fetch the first and the last task-IDs from the supplied time range using a secondary index ("creation DateTime"+"task Id"). And then to search "TaskSearchTable" by necessary filter words and by supplying start and end task-IDs as ExternalDocumentId**Start-Stop** limiting search range parameters.

[20170201]
In-memory + disk persistence. Storing resources synchronized between memory and a disk.

Sometimes it's necessary to have entities which must be stored inside of the in-memory dictionary, for the fastest access, and, at the same time, synchronized with the disk. Starting from DBreeze ver. 1.81 We have DBreezeEngineInstance.Resources, that is available right after DBreeze engine instantiation. It can be called for resource manipulation (Insert(Update)/Remove/Select) from any point of the program, inside or outside any transaction.

```
DBreezeEngine DBEngine = new DBreezeEngine("path to DBreeze folder or
configuration object");

DBEngine.Resources.Insert<MyResourceObjectType>("MyResourceName", new
MyResourceObjectType() { Property1 = "322223" });
```

```
var rsr = DBEngine.Resources.Select<MyResourceObjectType>("MyResourceName");

DBEngine.Resources.Remove("MyResourceName");

rsr = DBEngine.Resources.Select<MyResourceObjectType>("MyResourceName");
```

Resource identification is always "string", resource itself is any DBreeze convertible DataType or a DataType serializable by a supplied custom serializer (the same like value data type in casual e.g. DBreeze.Insert).

There are several function overloads letting us to work with the batches and extra technical settings regulating either resources must be held on-Disk or in-Memory, stored fast and with the insert validation check.

To support such functionality LTrie-File-Set will be created with the name "_DBreezeResources" inside of the DBreeze folder.

[20170202]
DBreezeEngine.Resources.SelectStartsWith.

```
DBEngine.Resources.Insert<int>("t1", 1);
            DBEngine.Resources.Insert<int>("t2", 2);
            DBEngine.Resources.Insert<int>("t3", 3);

            DBEngine.Resources.Insert<int>("b1", 1);
            DBEngine.Resources.Insert<int>("b2", 2);

            foreach (var xkw in DBEngine.Resources.SelectStartsWith<int>("t"))
            {
            Console.WriteLine(xkw.Key + ".." + xkw.Value);
            }

            foreach (var xkw in
DBEngine.Resources.SelectStartsWith<int>("b",new DBreezeResources.Settings {
SortingAscending = false}))
            {
            Console.WriteLine(xkw.Key + ".." + xkw.Value);
            }
```

[20170306]

InsertDataBlockWithFixedAddress

This new function from ver. 1.84, always returns a fixed address to the inserted data-block, even if it changes location in the file (e.g. after updates).

```csharp
using (var t = eng.GetTransaction())
            {
            byte[] blref = t.Select<int, byte[]>("t1", 1).Value;
            blref = t.InsertDataBlockWithFixedAddress<byte[]>("t1", blref, new
byte[] { 1, 2, 3 });
            t.Insert<int, byte[]>("t1", 1, blref);

            t.Commit();
            }

            using (var t = eng.GetTransaction())
            {
            byte[] blref = t.Select<int, byte[]>("t1", 1).Value;
            var vall = t.SelectDataBlockWithFixedAddress<byte[]>("t1", blref);

            }

using (var t = eng.GetTransaction())
            {
            byte[] blref = t.Select<int, byte[]>("t1", 1).Value;

                    blref = t.InsertDataBlockWithFixedAddress<byte[]>("t1",
blref, new byte[10000]);
            t.Insert<int, byte[]>("t1", 1, blref);

            t.Commit();
            }

      using (var t = eng.GetTransaction())
            {
            byte[] blref = t.Select<int, byte[]>("t1", 1).Value;
            var vall = t.SelectDataBlockWithFixedAddress<byte[]>("t1", blref);

            }

//Also possible:
 using (var t = eng.GetTransaction())
            {
            var row = t.Select<int, byte[]>("t1", 1);
            var vall = row.GetDataBlockWithFixedAddress<byte[]>(0);
            }
```

Several tips for the use-case and benchmarks.

Let's assume that we have an Entity and we want to have 2 extra secondary indexes to search this entity, we will use 1 table to store everything.

After byte[] {2} we will store the primary key - entity ID, after byte[] {5} - first secondary index, after byte[] {6} - second secondary index.
Value in all cases will be reference to the DataBlockWithFixedAddress.

After byte[] {1} we store monotonically grown id

```csharp
using DBreeze.Utils;

//Storing in dictionary first, to stay with sorted batch insert - fast speed,
smaller file size
Dictionary<string, Tuple<byte[], byte[]>> df = new Dictionary<string,
Tuple<byte[], byte[]>>();

byte[] ik = null;
byte[] ref2v = null;
```

INSERT

```csharp
using (var t = eng.GetTransaction())
{
//Getting initial ID
        int idx = t.Select<byte[], int>("t1", new byte[] { 1 }).Value + 1;
//Inserting 100K entities
            for (int i = 0; i < 100000; i++)
            {
            //Inserting datablock
                ref2v = t.InsertDataBlockWithFixedAddress<byte[]>("t1",
null, new byte[200]);

            //Inserting primary key where value is a pointer to a data
                ik = new byte[] { 2
}.Concat(idx.To_4_bytes_array_BigEndian());
                df.Add(ik.ToBytesString(), new Tuple<byte[], byte[]>(ik,
ref2v));

                    //Inserting first secondary index
                //After byte[] {5} must come byte[] associating with the key
 of the secondary index, instead of idx
                ik = new byte[] { 5
}.Concat(idx.To_4_bytes_array_BigEndian());
```

```
                df.Add(ik.ToBytesString(), new Tuple<byte[], byte[]>(ik,
ref2v));

                    ///Inserting second secondary index
                //After byte[] {6} must come byte[] associating with the key
of the secondary index, instead of idx
                ik = new byte[] { 6
}.Concat(idx.To_4_bytes_array_BigEndian());
                df.Add(ik.ToBytesString(), new Tuple<byte[], byte[]>(ik,
ref2v));

//index grows
                idx++;


            }
}

//Insert itself
 foreach (var el in df.OrderBy(r => r.Key))
        {
                t.Insert<byte[], byte[]>("t1", el.Value.Item1,
el.Value.Item2);
        }

//Storing latest maximal ID of the entity
t.Insert<byte[],int>("t1",new byte[] {1}, --idx);

t.Commit();
}
```

Benchmarking:
Standard HDD, inserted 100K elements with 1 primary key and 2 secondary indexes.
Table file size 30MB, consumed 5 seconds (around 5 inserts per row were used).

Update:

For **small amount of updates**:

//Getting reference to the key 5 via primary index and updating it. Of course, it's possible to get reference via secondary indexes also:

```
var reference = t.Select<byte[], byte[]>("t1", new byte[] { 2
}.Concat(((int)5).To_4_bytes_array_BigEndian())).Value;
//Updating the value
t.InsertDataBlockWithFixedAddress<byte[]>("t1", reference, new byte[215]);
```

In case if we want to **update a huge batch** and we are not satisfied with the speed of previous technique, we can follow such logic:

```csharp
//It's necessary to toggle
t.Technical_SetTable_OverwriteIsNotAllowed("t1");
 for (int i = 0; i < 100000; i++)
             {
           //Again writing on the new place
                  ref2v = t.InsertDataBlockWithFixedAddress<byte[]>("t1",
null, new byte[215]);

                  ik = new byte[] { 2
}.Concat(i.To_4_bytes_array_BigEndian());
                  df.Add(ik.ToBytesString(), new Tuple<byte[], byte[]>(ik,
ref2v));

                  //After byte[] {5} must come instead of idx - byte[]
associating with the key of the secondary index
                  ik = new byte[] { 5
}.Concat(i.To_4_bytes_array_BigEndian());
                  df.Add(ik.ToBytesString(), new Tuple<byte[], byte[]>(ik,
ref2v));

                  //After byte[] {6} must come instead of idx - byte[]
associating with the key of the secondary index
                  ik = new byte[] { 6
}.Concat(i.To_4_bytes_array_BigEndian());
                  df.Add(ik.ToBytesString(), new Tuple<byte[], byte[]>(ik,
ref2v));

             }

//Updating
      foreach (var el in df.OrderBy(r => r.Key))
           {
                  t.Insert<byte[], byte[]>("t1", el.Value.Item1,
el.Value.Item2);
             }
```

Benchmarking:
Standard HDD, update 100K elements with 1 primary key and 2 secondary indexes.
Table file size became 60MB from 30MB, consumed 8 seconds (around 5 inserts per row were consumed).

Random/Sequential selects

```
  Random rnd = new Random();
            for (int i = 0; i < 10000; i++)                 {
                    int k = rnd.Next(99999);

                    byte[] bt = t.Select<byte[], byte[]>("t1", new byte[] { 5
}.Concat(k.To_4_bytes_array_BigEndian()))
                            .GetDataBlockWithFixedAddress<byte[]>(0);
            }
```

Random **select becomes several times faster** in comparison with the case when the secondary index needs to lookup value via primary key. Inserts are faster and file size is smaller in comparison with the technique, when we store entities with each secondary index separately.

Benchmarking:
Standard HDD, **10K random** lookups takes **500ms**; **sequential** lookup takes **120ms**

New overloads:

We can **get DataBlockWithFixedAddress** content **via** DBreeze.DataTypes.**Row**, having that pointer to it is a part of row's value.

[20170319]
RandomKeySorter

Starting from ver. 1.84, instance of the transaction contains instantiated class RandomKeySorter. It can be very handy in case of **batch insert of random keys** or **batch update of random keys** with the flag "**Technical_SetTable_OverwriteIsNotAllowed**()". Huge speed increase and space economy can be achieved in such scenarios. We have discussed earlier many times that DBreeze is sharpened for the insert of sorted keys within huge batch operations, so here there is a useful wrapper.

```
 using (var t = eng.GetTransaction())
 {
     //RandomKeySorter is accessible via transaction

     //AutomaticFlushLimitQuantityPerTable by default is 10000
```

```
    //t.RandomKeySorter.AutomaticFlushLimitQuantityPerTable = 100000;
    //When the quantity of operations per table is more or equal to
AutomaticFlushLimitQuantityPerTable (or by committing the transaction),
operations will be executed in sorted (ascending by the key) manner.

    //First Remove operations, then Insert operations will be executed.


    Random rnd = new Random();
    int k = 0;
    HashSet<int> ex = new HashSet<int>();
    for (int i = 0; i < 2000; i++)
    {
            while (true)
            {
                    k = rnd.Next(3000000);
                if (!ex.Contains(k))
                    {
                            ex.Add(k);
                            break;
                    }
            }

            t.RandomKeySorter.Insert<int,byte[]>("t1", k, new byte[] { 1
});

            //Or remove
            //t.RandomKeySorter.Remove<int>("t1", 1);
    }

    //Automatic flushing entities from RandomKeySorted
        t.Commit();
    }
```

Note, while committing, keys will be removed first, then added

New overloads:

```
 t.RandomKeySorter.Insert = t.InsertRandomKeySorter
 t.RandomKeySorter.Remove = t.RemoveRandomKeySorter
```


[20170321]
DBreeze as an object database. Objects and Entities.

Starting from ver. 1.84, there is a new data storage concept available (only for new tables).

This approach will be interesting for the case when an entity/object has more than 1 search key (primary). System will automatically add and remove indexes within CRUD operations. Many DBreeze optimization concepts like "technical_SetTableOVerwrite" and "sorting keys in memory before insert" are already implemented inside of this software.

API explanation

Let's define the custom serializer for DBreeze (in this example let's take NetJSON from NuGet)

```csharp
using DBreeze;
using DBreeze.Utils;
using DBreeze.Objects;

 DBreezeEngine  eng = new DBreezeEngine(@"D:\Temp\x1");

DBreeze.Utils.CustomSerializator.ByteArraySerializator = (object o) => { return
NetJSON.NetJSON.Serialize(o).To_UTF8Bytes(); };

DBreeze.Utils.CustomSerializator.ByteArrayDeSerializator = (byte[] bt, Type t)
=> { return NetJSON.NetJSON.Deserialize(t,bt.UTF8_GetString()); };
```

Insert

Let's *insert* 1000 entities of type Person:

```csharp
public class Person
{
    public long Id { get; set; }
    public string Name { get; set; }
    public DateTime Birthday { get; set; }
    public decimal Salary { get; set; }
}
```

```csharp
DateTime initBirthday = new DateTime(1977, 10, 10);
Random rnd = new Random();
Person p = null;

using (var t = eng.GetTransaction())
{
    t.SynchronizeTables("t1"); //not needed if only 1 table is under
modification

for (int i = 1; i <= 1000; i++)
{
```

```csharp
    p = new Person
    {
        Id = t.ObjectGetNewIdentity<long>("t1"), //Automatic identity generator
//Identity will grow up monotonically

        Birthday = initBirthday.AddYears(rnd.Next(40)).AddDays(i),
        Name = $"Mr.{i}",
        Salary = 12000
    };

    var ir = t.ObjectInsert<Person>("t1", new DBreezeObject<Person>
    {
        Indexes = new List<DBreezeIndex>
        {
            new DBreezeIndex(1,p.Id) { PrimaryIndex = true }, //PI Primary
Index
//One PI must be set, if any secondary index will append it to the end, for
uniqueness

            new DBreezeIndex(2,p.Birthday), //SI - Secondary Index
            //new DBreezeIndex(3,p.Salary) //SI
            //new DBreezeIndex(4,p.Id) { AddPrimaryToTheEnd = false } //SI
        },

        NewEntity = true,
         //Changes Select-Insert pattern to Insert (speeds up insert process)

        Entity = p //Entity itself
    },
false);
//Last parameter must be set to true if we need higher speed of a CRUD operation
(will consume more physical space)

}

    t.Commit()
}
```

There are 3 new functions available via Transaction instance, they all start from the word
"Object":
```
ObjectGetNewIdentity
ObjectInsert
ObjectRemove
```
And one new function available via `DBreeze.DataTypes.Row.ObjectGet.`

There can be 255 indexes per entity (value 0 is reserved). Indexes numbers must be

specified within insert or select operations.

Entity itself is saved only once, then each index becomes reference to it.
This concept reduces space and speeds up updates.
All indexes are stored in one table starting from the byte defining this index (1,2,3 etc…).
Under byte 0 is a saved identity counter, that brings the function ObjectGetNewIdentity.

Last parameter of the function ObjectInsert must be set to true if the batch CRUD operation must be speeded up. Note that it can reside in more physical space.

Parameter **DBreezeObject.NewEntity** is set by the programmer. It helps the system to skip Select operation before Insert and can increase the insert speed of new entities and may be set **for new entities only**.

Insert / Update rules:

- If an entity is not changed it is not going to be saved (time economy).
- If the index is not supplied within update or not changed - it will not be saved (time economy).
- New index entries for the entity can be added within the update.
- If the index is changed then the old one will be removed and the new one will be inserted instead.
- Primary keys should be not changeable, because they can be added to the end of some secondary indexes (switched by DBreezeIndex.AddPrimaryToTheEnd = false)
- **ObjectInsert** returns DBreeze.Objects.**DBreezeObjectInsertResult** with different useful information.
- To delete entity's indexed parameter it must be supplied with **null** value e.g.
    new DBreeze.Objects.DBreezeIndex(2,null)   //Removes entity's property from index 2
- To get higher CRUD speed, the last parameter of ObjectInsert must be changed to true. It can consume more physical space

Selects / Retrieving data:

Getting single object stored under Primary Key 5:

```
 var exp = t.Select<byte[], byte[]>("t1", 1.ToIndex((long)5))
            .ObjectGet<Person>();
```

Variable exp is of type DBreezeObject.
To get e.g. entity's property Name: exp.Entity.Name

ToIndex is a function helping to create byte[]. In this case it will create byte[] from (byte)1 and (long)5. Where (byte)1 is the index identifier and 5 is the primary key.

Using DBreeze.Utils;

```
1.ToIndex((long)5)
= ((byte)1).To_1_byte_array().Concat(((long)5).To_8_bytes_array_BigEndian())
```

In the next example: (byte)2 + DateTime + (long) - that's how birthday index will be stored

```
2.ToIndex(new DateTime(2007, 10, 15), (long)5)
=
((byte)2).To_1_byte_array().ConcatMany((new DateTime(2007, 10,
15)).To_8_bytes_array(), ((long)5).To_8_bytes_array_BigEndian())
```

There is also another useful function for fast byte[] keys crafting:
DBreeze.Utils.ToBytes

```
Using DBreeze.Utils;

Console.WriteLine(127.ToBytes((long)45).ToBytesString());
```

Will result to: 0x8000007F800000000000002D
DBreeze bytes conversion leads to:
8000007F - 4 bytes equal to integer 127
800000000000002D - 8 bytes equal to long 45

.ToBytes vs .ToIndex
ToBytes differs from ToIndex by the way of first element casting. ToIndex tries to cast integer
as byte and ToBytes casts it as integer:

```
12.ToBytes().ToBytesString() = 0x 8000000C
12.ToIndex(new byte[] { 1 }).ToBytesString() = 0x 0C01
```

Getting range of objects via primary key (5-20)

```
foreach (var rw in t.SelectForwardFromTo<byte[], byte[]>("t1",
            1.ToIndex((long)5), true,
            1.ToIndex((long)20), true))
        {
            var tt = rw.ObjectGet<Person>();
            Console.WriteLine(tt.Entity.Id + "_" + tt.Entity.Name + "_"
+ tt.Entity.Birthday.ToString("dd.MM.yyyy"));
        }
```

Getting range of objects via secondary index stored under key 2 (birthdays). Note, that within

insert, by default, primary key will be added to the end of secondary key, so our search through birthdays will look like this:

```csharp
foreach (var rw in t.SelectForwardFromTo<byte[], byte[]>("t1",
                2.ToIndex(new DateTime(2007, 10, 15), long.MinValue), true,
                2.ToIndex(new DateTime(2007, 10, 25), long.MaxValue), true
        ))
        {
                var tt = rw.ObjectGet<Person>();
                Console.WriteLine(tt.Entity.Id + "_" + tt.Entity.Name + "_"
+ tt.Entity.Birthday.ToString("dd.MM.yyyy"));
        }
```

*In case if we want to use **StartFrom** (having no idea about the end) we use **dataType max/min** values: getting all persons born starting from 17 November 2007*

```csharp
foreach (var rw in t.SelectForwardFromTo<byte[], byte[]>("t1",
                2.ToIndex(new DateTime(2007, 10, 17), long.MinValue), true,
                2.ToIndex(DateTime.MaxValue, long.MaxValue), true
        ))
        {
                var tt = rw.ObjectGet<Person>();
                Console.WriteLine(tt.Entity.Id + "_" + tt.Entity.Name + "_"
+ tt.Entity.Birthday.ToString("dd.MM.yyyy"));
        }
```

Because all indexes are stored in one table, we have to use range limitations (read about DBreeze indexes) and **Forward/Backward FromTo** becomes this concept's favorite **function** for the range selects.

Update

Update can look like insert, but without "NewEntity=true" parameter:

```csharp
p = new Person
{
    Id = 15,
    Birthday = new DateTime(2007, 10, 12),
    Name = $"Mr.{i}",
    Salary = 12000
};

var ir = t.ObjectInsert<Person>("t1", new DBreeze.Objects.DBreezeObject<Person>
{
    Indexes = new List<DBreeze.Objects.DBreezeIndex>
    {
        new  DBreeze.Objects.DBreezeIndex(1,p.Id) { PrimaryIndex = true },
      //we need any available index to find the object in database
```

```
        new  DBreeze.Objects.DBreezeIndex(2, null),
        //Removing birthday-index for this person

 //we could supply all available indexes here, like in the first insert - it's
 typical behavior - (they will not be overwritten if they are not changed). But
 it's not necessary if we are sure that we don't want to change them

        //Adding new indexed parameter (only for this entity, not for all)
        new DBreezeIndex(3,p.Salary), //SI - Secondary Index

    },
    //NewEntity = true,
    Entity = p
}, false);
```

But very often it's necessary to get data from the database first, change it and then save back. Second possibility of update:

```
//Getting entity from database
var ex = t.Select<byte[], byte[]>("t1", 1.ToIndex((long)i))
    .ObjectGet<Person>();

//Updating entity
ex.Entity.Name = "Superman";

//Setting changed indexes, if needed (other will stay if not supplied)
ex.Indexes = new List<DBreezeIndex>
{
    new DBreezeIndex(1,ex.Entity.Id) { PrimaryIndex = true },
    new DBreezeIndex(2,ex.Entity.Birthday),
};

//Saving entity
var ir = t.ObjectInsert<Person>("t1", ex,
    true); //With e.g. high-speed
```

Remove entity

To remove entity we need to supply at least one of the indexes:

```
 foreach (var rw in t.SelectForwardFromTo<byte[], byte[]>("t1",
              2.ToIndex(new DateTime(2007, 10, 17), long.MinValue), true,
```

```
                2.ToIndex(new DateTime(2007, 10, 19), long.MaxValue), true
        ))
        {
                var tt = rw.ObjectGet<Person>();
                t.ObjectRemove("t1", 1.ToIndex(tt.Entity.Id));
            //or just  t.ObjectRemove("t1", rw.Key);
        }

        t.Commit();
```

ObjectGetNewIdentity - getting grown up identities for the table.

It's possible to create many counters which will be automatically stored in the table.
The default counter is stored under key address byte[] {0}. For other counters must be set address and, if it's needed, the seed.

```
Var p = new Person
{
    Id = t.ObjectGetNewIdentity<long>("t1"),
    Birthday = initBirthday,
    Name = "Wagner",
    Salary = 12000,
    ExtraIdentity = t.ObjectGetNewIdentity<long>("t1", new byte[] { 255, 1 },4)
};
```

Here ExtraIdentity will be stored under key address new byte[] {255,1}. So this place is reserved for this counter with the seed 4 (4, 8, 12, 16 etc...).

Creating another one with the seed 7 (7, 14, 21, 28...):
```
ExtraIdentity = t.ObjectGetNewIdentity<ulong>("t1", new byte[] { 255, 2 },7)
```

Creating another one with the seed 3 (3, 6, 9, 12...):
```
ExtraIdentity = t.ObjectGetNewIdentity<int>("t1", new byte[] { 255, 3 },3)
```

In this case **"t1" can not use index 255 for inserting objects**, because it's already **busy** with the user's key generation.

ObjectGetByFixedAddress gives ability access DBreeze object directly by pointer

```
//Inserting
            byte[] ptr1 = null;
```

```csharp
            byte[] ptr2 = null;

            using (var tran = engine.GetTransaction())
            {
                var x1 = tran.ObjectInsert<byte[]>("t1", new
DBreeze.Objects.DBreezeObject<byte[]>()
                    {
                        Entity = new byte[] { 1, 2, 3 },
                        NewEntity = true,
                        Indexes = new List<DBreeze.Objects.DBreezeIndex>()
                          {
                                new DBreeze.Objects.DBreezeIndex(1, (long)1){ PrimaryIndex =
true }
                          }
                    });

                ptr1 = x1.PtrToObject;

                x1 = tran.ObjectInsert<byte[]>("t1", new
DBreeze.Objects.DBreezeObject<byte[]>()
                    {
                        Entity = new byte[] { 2, 6, 3 },
                        NewEntity = true,
                        Indexes = new List<DBreeze.Objects.DBreezeIndex>()
                          {
                                new DBreeze.Objects.DBreezeIndex(1, (long)2){ PrimaryIndex =
true }
                          }
                    });

                ptr2 = x1.PtrToObject;

                tran.Commit();
            }

            //Getting
            using (var tran = engine.GetTransaction())
            {
                //Variant 1 via index
                var exp = tran.Select<byte[], byte[]>("t1", 1.ToIndex((long)1))
                    .ObjectGet<byte[]>();

                //via pointer
                var do1 = tran.ObjectGetByFixedAddress<byte[]>("t1", ptr1);
                Console.WriteLine(do1.Entity.ToBytesString());

                do1 = tran.ObjectGetByFixedAddress<byte[]>("t1", ptr2);
                Console.WriteLine(do1.Entity.ToBytesString());
            }
```

Explanation of update entity strategies

```
//Inserting new entity
        using (var tran = engine.GetTransaction())
        {
            var x1 = tran.ObjectInsert<byte[]>("t1", new
DBreeze.Objects.DBreezeObject<byte[]>()
            {
                Entity = new byte[] { 1, 2, 3 },
                //NewEntity = true, //this is another speed optimization flag, can
be skipped until is really necessary
                Indexes = new List<DBreeze.Objects.DBreezeIndex>()
                    {
                        new DBreeze.Objects.DBreezeIndex(1, (long)1){ PrimaryIndex =
true }
                    }
            }, false); //inserting entity

            tran.Commit();

        }

        //Testing updates
        using (var tran = engine.GetTransaction())
        {


            var x1 = tran.ObjectInsert<byte[]>("t1", new
DBreeze.Objects.DBreezeObject<byte[]>()
            {
                Entity = new byte[] { 1, 2, 3 },
                Indexes = new List<DBreeze.Objects.DBreezeIndex>()
                    {
                        new DBreeze.Objects.DBreezeIndex(1, (long)1){ PrimaryIndex =
true }
                    }
            }, false);  //Speed insert is off, the equal entity on the level of
byte[] will not be overwritten

            //--> will be false
            Console.Write(x1.EntityWasInserted);


            x1 = tran.ObjectInsert<byte[]>("t1", new
DBreeze.Objects.DBreezeObject<byte[]>()
            {
                Entity = new byte[] { 1, 2, 3 },
                Indexes = new List<DBreeze.Objects.DBreezeIndex>()
                    {
                        new DBreeze.Objects.DBreezeIndex(1, (long)1){ PrimaryIndex =
true }
                    }
            }, true); //Speed insert is on - entity will be overwritten (new
PtToObject will be generated)
```

```
            //--> will be true
            Console.Write(x1.EntityWasInserted);


            x1 = tran.ObjectInsert<byte[]>("t1", new
DBreeze.Objects.DBreezeObject<byte[]>()
            {
                Entity = new byte[] { 1, 2, 3, 4 },
                Indexes = new List<DBreeze.Objects.DBreezeIndex>()
                {
                        new DBreeze.Objects.DBreezeIndex(1, (long)1){ PrimaryIndex =
true }
                }
            }, false);  //Speed insert is off

            //--> will be true because entity has changed
            Console.Write(x1.EntityWasInserted);

            tran.Commit();

        }
```
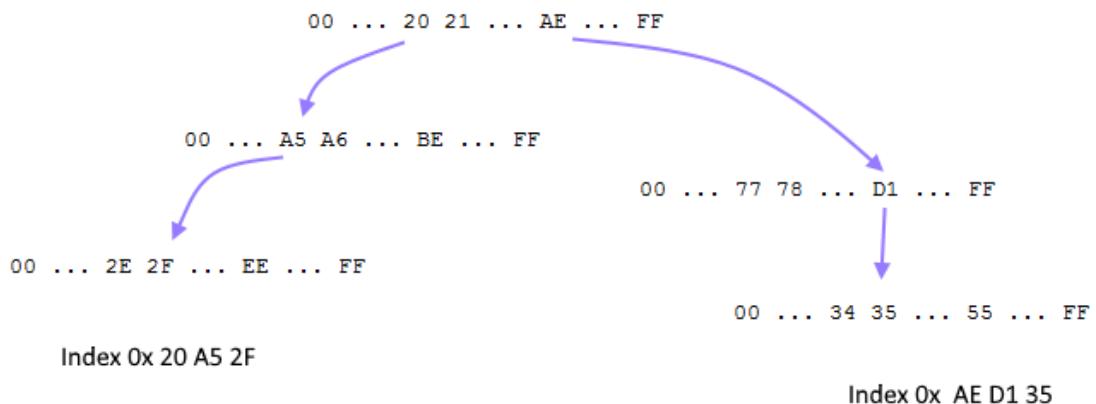
[20170327]

How DBreeze index works.

Index 0x 20 A5 2F

Index 0x AE D1 35

DBreeze table keys are lexicographically sorted

```
using DBreeze;
using DBreeze.Utils;


using (var t = engine.GetTransaction())
```

```csharp
{
    t.Insert<string, byte[]>("t1", "a", null);
    t.Insert<string, byte[]>("t1", "aa", null);
    t.Insert<string, byte[]>("t1", "aaa", null);
    t.Insert<string, byte[]>("t1", "aab", null);
    t.Insert<string, byte[]>("t1", "aac", null);
    t.Insert<string, byte[]>("t1", "aad", null);
    t.Insert<string, byte[]>("t1", "c", null);
    t.Insert<string, byte[]>("t1", "cc", null);
    t.Insert<string, byte[]>("t1", "cca", null);
    t.Insert<string, byte[]>("t1", "ccb", null);
    t.Insert<string, byte[]>("t1", "ccc", null);
    t.Insert<string, byte[]>("t1", "ccd", null);
    t.Insert<string, byte[]>("t1", "b", null);
    t.Insert<string, byte[]>("t1", "bb", null);
    t.Insert<string, byte[]>("t1", "bba", null);
    t.Insert<string, byte[]>("t1", "bbb", null);
    t.Insert<string, byte[]>("t1", "bbc", null);
    t.Insert<string, byte[]>("t1", "bbd", null);


    t.Commit();
}

using (var t = engine.GetTransaction())
{
    foreach (var r in t.SelectForward<string, byte[]>("t1"))
    {
        Console.WriteLine(r.Key);
    }
    /*
a
aa
aaa
aab
aac
aad
b
bb
bba
bbb
bbc
bbd
c
cc
cca
ccb
ccc
ccd
```

```csharp
        */

        foreach (var r in t.SelectForwardFromTo<string, byte[]>("t1","aab",true,
"aad",true))
        {
            Console.WriteLine(r.Key);
        }

        /*
aab
aac
aad
        */


        foreach (var r in t.SelectForwardFromTo<string, byte[]>("t1", "aa", true,
"bb", true))
        {
            Console.WriteLine(r.Key);
        }
        /*
        aa
        aaa
        aab
        aac
        aad
        b
        bb
        */

        foreach (var r in t.SelectForwardStartsWith<string, byte[]>("t1", "bb"))
        {
            Console.WriteLine(r.Key);
        }
        /*
bb
bba
bbb
bbc
bbd
         */


        foreach (var r in t.SelectForwardFromTo<string, byte[]>("t1", "a", true,
"b", false))
        {
            Console.WriteLine(r.Key);
        }
        /*
a
```

```
aa
aaa
aab
aac
aad
    */

    foreach (var r in t.SelectForwardFromTo<string, byte[]>("t1", "aa", true,
"b", false))
    {
        Console.WriteLine(r.Key);
    }
    /*
aa
aaa
aab
aac
aad
    */
}
```

DBreeze numerical to byte[] conversion *functions are built up in such a way that **smaller values** of the same type are always **lexicographically earlier than bigger** values of the same type.*

```
using (var t = engine.GetTransaction())
{
    t.Insert<int, byte[]>("t1", int.MinValue, null);
    t.Insert<int, byte[]>("t1", int.MaxValue, null);
    t.Insert<int, byte[]>("t1", -10, null);
    t.Insert<int, byte[]>("t1", -16, null);
    t.Insert<int, byte[]>("t1", 0, null);
    t.Insert<int, byte[]>("t1", 16, null);
    t.Insert<int, byte[]>("t1", 10, null);


    t.Insert<long, byte[]>("t2", long.MinValue, null);
    t.Insert<long, byte[]>("t2", long.MaxValue, null);
    t.Insert<long, byte[]>("t2", -10, null);
    t.Insert<long, byte[]>("t2", -16, null);
    t.Insert<long, byte[]>("t2", 0, null);
    t.Insert<long, byte[]>("t2", 16, null);
    t.Insert<long, byte[]>("t2", 10, null);
```

```csharp
        t.Insert<DateTime, byte[]>("t3", DateTime.MinValue, null);
        t.Insert<DateTime, byte[]>("t3", DateTime.MaxValue, null);
        t.Insert<DateTime, byte[]>("t3", new DateTime(2017, 5, 1), null);
        t.Insert<DateTime, byte[]>("t3", new DateTime(2016, 4, 1), null);
        t.Insert<DateTime, byte[]>("t3", new DateTime(2017, 9, 1), null);
        t.Insert<DateTime, byte[]>("t3", new DateTime(2018, 3, 1), null);


        t.Commit();
}

using (var t = engine.GetTransaction())
{

        foreach (var r in t.SelectForward<int, byte[]>("t1"))
        {
            Console.WriteLine(r.Key.To_4_bytes_array_BigEndian().ToBytesString() + "
        " + r.Key);
        }
/*
00000000      -2147483648
7FFFFFF0      -16
7FFFFFF6      -10
80000000      0
8000000A      10
80000010      16
FFFFFFFF      2147483647
 */

        foreach (var r in t.SelectForward<long, byte[]>("t2"))
        {
            Console.WriteLine(r.Key.To_8_bytes_array_BigEndian().ToBytesString() + "
        " + r.Key);
        }
/*
0000000000000000      -9223372036854775808
7FFFFFFFFFFFFFF0      -16
7FFFFFFFFFFFFFF6      -10
8000000000000000      0
800000000000000A      10
8000000000000010      16
FFFFFFFFFFFFFFFF      9223372036854775807
 */

        foreach (var r in t.SelectForward<DateTime, byte[]>("t3"))
        {
            Console.WriteLine(r.Key.To_8_bytes_array().ToBytesString() + "    " +
r.Key.ToString("yyyy/MM/dd"));
        }
```

```
/*
0000000000000000     0001.01.01
08D359C0918E8000     2016.04.01
08D4902502B9C000     2017.05.01
08D4F0CC63890000     2017.09.01
08D57F07604DC000     2018.03.01
2BCA2875F4373FFF     9999.12.31
*/
}
```

Creating a complex key that contains several data types inside:

```csharp
using (var t = engine.GetTransaction())
{
    byte[] key = null;

    key = new byte[] { 1 }
                .ConcatMany(
                        new DateTime(2017, 5, 1).To_8_bytes_array(),
                        ((long)125).To_8_bytes_array_BigEndian()
                        );

    t.Insert<byte[], byte[]>("t1", key, null);

    key = new byte[] { 1 }
                .ConcatMany(
                        new DateTime(2017, 5, 1).To_8_bytes_array(),
                        ((long)126).To_8_bytes_array_BigEndian()
                        );

    t.Insert<byte[], byte[]>("t1", key, null);

    key = new byte[] { 1 }
                .ConcatMany(
                        new DateTime(2017, 5, 1).To_8_bytes_array(),
                        ((long)127).To_8_bytes_array_BigEndian()
                        );

    t.Insert<byte[], byte[]>("t1", key, null);

    t.Commit();
}


using (var t = engine.GetTransaction())
{
    foreach (var r in t.SelectForward<byte[], byte[]>("t1"))
    {
```

```
            Console.WriteLine(
                    r.Key.Substring(0, 1).ToBytesString() + " " +
                    r.Key.Substring(1, 8).ToBytesString() + " " +
                    r.Key.Substring(9, 8).ToBytesString() +
                    "  ->  " +
                    r.Key.Substring(0,1).To_Byte().ToString() + " " +
                    r.Key.Substring(1, 8).To_DateTime().ToString("yyyy/MM/dd") + " " +
                    r.Key.Substring(9, 8).To_Int64_BigEndian().ToString() + " "
                    );
        }
    }

    /*
    01 08D4902502B9C000 800000000000007D  ->  1 2017.05.01 125
    01 08D4902502B9C000 800000000000007E  ->  1 2017.05.01 126
    01 08D4902502B9C000 800000000000007F  ->  1 2017.05.01 127
     */
```

Optimizing code using ToIndex or ToBytes functions (putting inside the previous example insert)

```
key = 2.ToIndex(new DateTime(2017, 5, 1), (long)113);
t.Insert<byte[], byte[]>("t1", key, null);

key = 2.ToIndex(new DateTime(2017, 5, 1), (long)117);
t.Insert<byte[], byte[]>("t1", key, null);

key = 2.ToIndex(new DateTime(2017, 5, 1), (long)115);
t.Insert<byte[], byte[]>("t1", key, null);


/*
01 08D4902502B9C000 800000000000007D  ->  1 2017.05.01 125
01 08D4902502B9C000 800000000000007E  ->  1 2017.05.01 126
01 08D4902502B9C000 800000000000007F  ->  1 2017.05.01 127
02 08D4902502B9C000 8000000000000071  ->  2 2017.05.01 113
02 08D4902502B9C000 8000000000000073  ->  2 2017.05.01 115
02 08D4902502B9C000 8000000000000075  ->  2 2017.05.01 117
*/
```

Storing in one table 2 search indexes of one entity:

```
byte[] key = null;
using (var t = engine.GetTransaction())
{
    //Under index Nr 1 - the primary key is stored. It's unique.
    //e.g. EntityId
```

```csharp
    key = 1.ToIndex((long)1);
    t.Insert<byte[], byte[]>("t1", key, null);

    key = 1.ToIndex((long)2);
    t.Insert<byte[], byte[]>("t1", key, null);

    key = 1.ToIndex((long)3);
    t.Insert<byte[], byte[]>("t1", key, null);

    key = 1.ToIndex((long)4);
    t.Insert<byte[], byte[]>("t1", key, null);

/*
  Under index Nr 2 is stored entity creation date, so we could search over it
also.
  We have to add entity primary key to the end of this index to avoid key
overwriting   in case, when different entities have the same creation date.
*/

    key = 2.ToIndex(new DateTime(2017, 5, 1), (long)1);
    t.Insert<byte[], byte[]>("t1", key, null);

    key = 2.ToIndex(new DateTime(2017, 8, 1), (long)2);
    t.Insert<byte[], byte[]>("t1", key, null);

    key = 2.ToIndex(new DateTime(2017, 8, 1), (long)3);
    t.Insert<byte[], byte[]>("t1", key, null);

    //Entities with id 2 and 3 have similar creation date

    key = 2.ToIndex(new DateTime(2017, 10, 1), (long)4);
    t.Insert<byte[], byte[]>("t1", key, null);

    t.Commit();
}


//Showing table content after insert. All indexes are stored together.
using (var t = engine.GetTransaction())
{
    foreach (var r in t.SelectForward<byte[], byte[]>("t1"))
    {
        Console.WriteLine(
                r.Key.Substring(0, 1).ToBytesString() + " " +
                r.Key.Substring(1).ToBytesString()
                );
    }
}
/*
```

```
01 8000000000000001
01 8000000000000002
01 8000000000000003
01 8000000000000004
02 08D4902502B9C0008000000000000001
02 08D4D87040BAC0008000000000000002
02 08D4D87040BAC0008000000000000003
02 08D5085F5BED8000800000000000004
*/

//Showing entities which were created in the range (2017, 8, 1) - (2017, 11, 1)
using (var t = engine.GetTransaction())
{
    foreach (var r in t.SelectForwardFromTo<byte[], byte[]>
        ("t1",
            2.ToIndex(new DateTime(2017, 8, 1)),true,
            2.ToIndex(new DateTime(2017, 11, 1)), true
            ))
    {
        Console.WriteLine(
            r.Key.Substring(0, 1).ToBytesString() + " " +
            r.Key.Substring(1).ToBytesString() +
            "  ->  " +
            " entityCreatedOn: " + r.Key.Substring(1,
8).To_DateTime().ToString("yyyy/MM/dd") + " " +
            " entityId: " + r.Key.Substring(9, 8).To_Int64_BigEndian()
            );
    }
}

/*
02 08D4D87040BAC0008000000000000002  ->   entityCreatedOn: 2017.08.01   entityId:
2
02 08D4D87040BAC0008000000000000003  ->   entityCreatedOn: 2017.08.01   entityId:
3
02 08D5085F5BED8000800000000000004  ->   entityCreatedOn: 2017.10.01   entityId:
4
 */
```

[20170330]

Simple DBreeze operations

```
using DBreeze;
using DBreeze.Utils;

using (var t = engine.GetTransaction())
{
```

```csharp
    t.SynchronizeTables("t1", "t2", "t3");

    t.Insert<int, byte[]>("t1", t.ObjectGetNewIdentity<int>("t1"), new byte[] {
1, 2, 3 });
    t.Insert<int, byte[]>("t1", t.ObjectGetNewIdentity<int>("t1"), new byte[] {
4, 5, 6 });


    Person p = new Person { Id = t.ObjectGetNewIdentity<long>("t2"), Birthday =
new DateTime(1970, 1, 1), Name = "Steven" };
    t.Insert<long, Person>("t2", p.Id, p);
    p = new Person { Id = t.ObjectGetNewIdentity<long>("t2"), Birthday = new
DateTime(1952, 4, 10), Name = "Seagal" };
    t.Insert<long, Person>("t2", p.Id, p);

    t.Insert<string, Guid>("t3", "First", Guid.NewGuid());
    t.Insert<string, Guid>("t3", "Second", Guid.NewGuid());

    //Committing transaction (Rollback automatic)
    t.Commit();
}



using (var t = engine.GetTransaction())
{
    foreach (var r in t.SelectForward<byte[], byte[]>("t1"))
    {
       Console.WriteLine(
            r.Key.ToBytesString()
            );
    }
    /*
00    - here will be stored identity
80000001
80000002
    */


    foreach (var r in t.SelectForwardStartFrom<byte[],
byte[]>("t1",2.ToBytes(),true))
    {
       Console.WriteLine(
            r.Key.ToBytesString()
            );
    }
/*
80000002
*/
```

```csharp
    foreach (var r in t.SelectForwardFromTo<byte[], byte[]>("t1", 1.ToBytes(),
true, 2.ToBytes(), true))
    {
        Console.WriteLine(
            r.Key.ToBytesString()
            );
    }
/*
80000001
80000002
 */


    foreach (var r in t.SelectForward<byte[], byte[]>("t2"))
    {
        Console.WriteLine(
            r.Key.ToBytesString()
            );
    }
/*
00    - here will be stored identity
8000000000000001
8000000000000002
*/

    foreach (var r in t.SelectForwardStartFrom<long, Person>("t2",1,true))
    {
        Console.WriteLine(
            r.Key + "    " + r.Value.Name
            );
    }
/*
1   Steven
2   Seagal
*/

    foreach (var r in t.SelectForward<string, byte[]>("t3"))
    {
        Console.WriteLine(
            r.Key
            );
    }
/*
First
Second
*/

    var row = t.Select<long, Person>("t2", 2);
    if (row.Exists)
        Console.WriteLine(row.Value.Birthday.ToString());
```

```
    /*
    10.04.1952 0:00:00
    */
    }
```

Select-Insert

```csharp
using (var t = engine.GetTransaction())
{
    foreach (var r in t.SelectForwardStartFrom<byte[], byte[]>("t1",
2.ToBytes(), true,
        true)) //ReadCursor
    {
        //Update
        t.Insert<byte[], byte[]>("t1", r.Key, new byte[] { 7, 7, 7, 7 });

    }

    t.Commit();
}

using (var t = engine.GetTransaction())
{
    foreach (var r in t.SelectForward<byte[], byte[]>("t1"))
    {
        Console.WriteLine(
            r.Key.ToBytesString() + " " + r.Value.ToBytesString()
            );
    }
}
/*
Key        Value        Description
00         80000002      - Identity
80000001 010203        - Under key 1 of type int we store new byte[] {1,2,3}
80000002 07070707       - Updated value
*/
```

…

[20170522]
Parallel reads inside of one transaction, to get benefits of the .NET TPL (Task Parallel Library).

Starting from ver. 1.86 It's possible to make parallel queries on the same or different tables
(it's possible to run in parallel READ commands ONLY).
Note, that the read cursor will be automatically converted to return values into the

"ReadVisibilityScope" mode (changes made inside the transaction will not be reflected in returned values).

```csharp
using (var t = eng.GetTransaction())
        {
        t.Insert<int, int>("t1", 1, 1);
        t.Insert<int, int>("t1", 5, 5);
        t.Insert<int, int>("t1", 6, 6);
        t.Insert<int, int>("t2", 2, 2);

        t.TextInsert("tS", new byte[] { 1 }, "very well bella");
        t.TextInsert("tS", new byte[] { 3 }, "well concerned");
        t.TextInsert("tS", new byte[] { 2 }, "monthy");
        t.Commit();
        }



        using (var t = eng.GetTransaction())
        {
        t.Insert<int, int>("t1", 1, 3); //Test to change value without
commit
        t.Insert<int, int>("t2", 2, 3); //Test to change value without
commit

        List<byte[]> l1;
        List<byte[]> l2;

        Task.Run(() =>
        {
                Thread.Sleep(3000);
                var r1 = t.Select<int, int>("t1", 1);          //After 3
seconds waiting time will fail because transaction doesn't exist anymore
                if (r1.Exists)
                        Console.WriteLine("T1: " + r1.Value);
                else
                        Console.WriteLine("T1 not available");
        });


        Task.WaitAll(
                Task.Run(() => {
                        var r1 = t.Select<int,int>("t1", 1);           //Will
result 1

                        if (r1.Exists)
                        Console.WriteLine("T1: " + r1.Value);
                        else
                        Console.WriteLine("T1 not available");
                })
                ,
```

```csharp
                    Task.Run(() => {
                            var r1 = t.Select<int, int>("t1", 1);  //Will result
1
                            if (r1.Exists)
                            Console.WriteLine("T1: " + r1.Value);
                            else
                            Console.WriteLine("T1 not available");
                    })
                    ,
                    Task.Run(() => {

                            foreach (var el in t.SelectForward<int, int>("t1"))
//Will result 1,5,6

                            {
                            Console.WriteLine("it t1: " + el.Value);
                            }

                    })
                    ,
                    Task.Run(() =>
                    {
                            var r2 = t.Select<int, int>("t2", 2); //Will result 2
                            if (r2.Exists)
                            Console.WriteLine("T2: " + r2.Value);
                            else
                            Console.WriteLine("T2 not available");
                    })
                    ,
                    Task.Run(() =>
                    {
                            l1 =
t.TextSearch("tS").Block("well").GetDocumentIDs().ToList(); //Will result 01,03
                            foreach (var el in l1)
                            {
                            Console.WriteLine("tS well: " + el.ToBytesString());
                            }
                    })
                    ,
                    Task.Run(() =>
                    {
                            l2 =
t.TextSearch("tS").Block("monthy").GetDocumentIDs().ToList(); //Will result 02
                            foreach (var el in l2)
                            {
                            Console.WriteLine("tS mon: " + el.ToBytesString());
                            }
                    })
            );

            }
```

[20170621]

TextSearch with ignoring FullMatch and Contains search block by parameter ignoreOnEmptyParameters = true

Insert example:

```
using (var t = eng.GetTransaction())
{
    t.TextInsert("transtext", new byte[] { 1 }, "Apple Banana", "#LG1 #LG2");
    t.TextInsert("transtext", new byte[] { 2 }, "Banana MANGO", "#LG2 #LG3");
    t.Commit();
}
```

Starting from ver. 1.88 we can query like this:

```
using (var t = eng.GetTransaction())
{
    var block = t.TextSearch("transtext").BlockAnd("Apple",
ignoreOnEmptyParameters: true).And("", "", false, ignoreOnEmptyParameters:
true);
    Console.WriteLine($"---- {block.GetDocumentIDs().Count()} ---");//returns 1

    block = t.TextSearch("transtext").BlockAnd("Apple", ignoreOnEmptyParameters:
true).And("", "", false, ignoreOnEmptyParameters: false);
    Console.WriteLine($"---- {block.GetDocumentIDs().Count()} ---");//returns 0

    block = t.TextSearch("transtext").BlockAnd("Apple", ignoreOnEmptyParameters:
true).And("", "#LG1", false, ignoreOnEmptyParameters: true);
    Console.WriteLine($"---- {block.GetDocumentIDs().Count()} ---");//returns 1

    block = t.TextSearch("transtext").BlockAnd("Apple", ignoreOnEmptyParameters:
true).And("", "#LG3", false, ignoreOnEmptyParameters: true);
    Console.WriteLine($"---- {block.GetDocumentIDs().Count()} ---");//returns 0

    block = t.TextSearch("transtext").BlockAnd("", ignoreOnEmptyParameters:
true).And("", "#LG2", false, ignoreOnEmptyParameters: true);
    Console.WriteLine($"---- {block.GetDocumentIDs().Count()} ---");//returns 2

    block = t.TextSearch("transtext").BlockAnd("", ignoreOnEmptyParameters:
false).And("", "#LG2", false, ignoreOnEmptyParameters: true);
    Console.WriteLine($"---- {block.GetDocumentIDs().Count()} ---");//returns 0

}
```

Such format gives us flexibility in construction of complex search logic, in case some parameters should be ignored when they are empty.

[20171017] Multi-parameter search. Example of cascade TextSearch Exclude command.

```csharp
using DBreeze;
using DBreeze.Utils;
using System.Diagnostics;

DBreezeEngine eng = new DBreezeEngine(@"D:\Temp\DBR1");
            using (var tran = eng.GetTransaction())
            {
                tran.TextInsert("txt", new byte[] { 1 }, "entity_1", "#GR_14
#U_RU #U_EN #U_DE");
                tran.TextInsert("txt", new byte[] { 2 }, "entity_2", "#GR_14
#U_RU #U_EN #U_DE");
                tran.TextInsert("txt", new byte[] { 3 }, "entity_3", "#GR_14
#U_RU #U_EN #U_DE");

                tran.TextInsert("txt", new byte[] { 4 }, "entity_4", "#GR_15
#U_RU #U_EN #U_DE");
                tran.TextInsert("txt", new byte[] { 5 }, "entity_5", "#GR_15
#U_RU #U_EN #U_DE");
                tran.TextInsert("txt", new byte[] { 6 }, "entity_6", "#GR_15
#U_RU #U_EN #U_DE");

                tran.TextInsert("txt", new byte[] { 7 }, "entity_7", "#GR_16
#U_RU #U_EN #U_DE");
                tran.TextInsert("txt", new byte[] { 8 }, "entity_8", "#GR_16
#U_RU #U_EN #U_DE");
                tran.TextInsert("txt", new byte[] { 9 }, "entity_9", "#GR_16
#U_RU #U_EN #U_DE");

                tran.Commit();
            }

            using (var tran = eng.GetTransaction())
            {
                var ts = tran.TextSearch("txt");
                //TEST 1
                //var q = ts.Block("entity_4");
                //foreach (var el in q.GetDocumentIDs())
                //{
                //    Debug.WriteLine(el.ToBytesString());
                //}

                //TEST 2
```

```
                // Show all entities with tag "#U_DE" (Returns 9 items)
                //var q = ts.Block("", "#U_DE");
                //foreach (var el in q.GetDocumentIDs().Take(1000))
                //{
                //    Debug.WriteLine(el.ToBytesString());
                //}

                //TEST 3
                //Show all entities with tag "#U_DE" except entities having ANY
 OF "#GR_15", "#GR_16" tags (Returns 3 items)

                List<string> excludingTags = new List<string> { "#GR_15",
 "#GR_16" };

                var q = ts.Block("", "#U_DE");
                foreach(var egr in excludingTags)
                {
                    q = q.Exclude("", egr);
                }
                foreach (var el in q.GetDocumentIDs().Take(1000))
                {
                    Debug.WriteLine(el.ToBytesString());
                }
            }
```

[20180220] Integrated Binary and JSON serializer Biser.NET

The complete documentation is available on Biser.NET. This serializer is integrated into DBreeze and available via DBreeze.Utils. Here is a **quick start guide**:

- Grab from NuGet **Biser** (or **DBreeze** that contains Biser), grab from Nuget **BiserObjectify.**
- Let's assume you have several objects to serialize. It is necessary to prepare them.

Call the next line to create the code for the serializer:

```
var resbof = BiserObjectify.Generator.Run(typeof(TS6),true, @"D:\Temp\1\",
```

```
forBiserBinary: true, forBiserJson: true, exclusions: null, generateForDBreeze:
true);
```
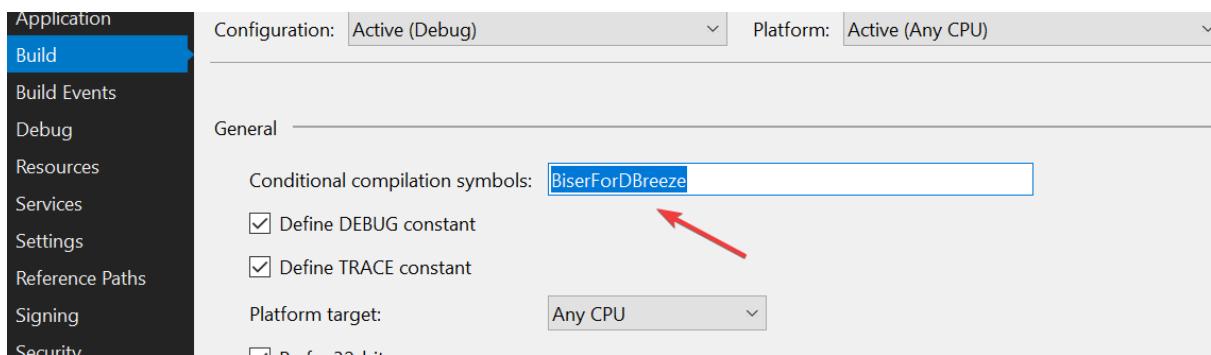
First argument is the type of the root object to be serialized (it can contain other objects that also must be serialized).

Second argument means that BiserObjectify must prepare a serializer for all objects included into the root object.

Third argument points to the folder where C# files for the serialization of each object will be created.

The fourth and fifth arguments mean that we want to use both Binary and JSON serializers.

The sixth argument is a HashSet (or null) with the property names that will not be serialized.

The seventh argument means that objects will be used by Biser integrated into DBreeze and some DBreeze necessary namespaces will be added when TRUE. Or it can be left FALSE (by default) and the object can be adjusted by adding "BiserForDBreeze" in "Conditional Compilation Symbols" of the project when used by DBreeze:

resbof variable will contain the same information that in generated files also as Dictionary.

- Copy generated files into your project and embed/link them to the project. Try to recompile.
- Probably, it will be necessary to add a "partial" keyword to objects that must be serialized. Compiler will warn you.

```
public partial class TS6
    {
        public string P1 { get; set; }
...
```

- Remove **BiserObjectify** from your project, it will not be necessary until next time.
- Usage:

```
TS6 t6 = new TS6()
```

```csharp
{
    P1 = "dsfs",
    P2 = 456,
    P3 = DateTime.UtcNow,
    P1 = "dsfs",
    P2 = 456,
    P3 = DateTime.UtcNow,
    P4 = new List<Dictionary<DateTime, Tuple<int, string>>>
        {
            new Dictionary<DateTime, Tuple<int, string>>{
                { DateTime.UtcNow.AddMinutes(-1), new
Tuple<int, string>(12,"testvar") },
                { DateTime.UtcNow.AddMinutes(-2), new
Tuple<int, string>(125,"testvar123") }
            },
            new Dictionary<DateTime, Tuple<int, string>>{
                { DateTime.UtcNow.AddMinutes(-3), new
Tuple<int, string>(17,"dsfsdtestvar") },
                { DateTime.UtcNow.AddMinutes(-4), new
Tuple<int, string>(15625,"sdfsdtestvar") }
            }
        },
    P5 = new Dictionary<int, Tuple<int, string>> {
        { 12, new Tuple<int, string>(478,"dsffdf") },
        { 178, new Tuple<int, string>(5687,"sdfsd") }
     },
    P6 = new Tuple<int, string, Tuple<List<string>,
DateTime>>(445, "dsfdfgfgfg",
        new Tuple<List<string>, DateTime>(new List<string> {
"a1", "a2" }, DateTime.Now.AddDays(58))),
    P7 = new List<string> { "fgdfgrdfg", "dfgfdgdfg" },
    P8 = new Dictionary<int, List<string>> {
            { 34,new List<string> { "drtttz","ghhtht"} },
            { 4534,new List<string> {
"dfgfghfgz","6546ghhtht"} }
        },

P25 = new Dictionary<int, List<string[,][][,,]>>[,,,][][,,]

...
}
```

Binary serialization:

```
var serializedObjectAsByteArray = t6.BiserEncoder().Encode();
var retoredBinaryObject= TS6.BiserDecode(serializedObjectAsByteArray);
```

NOTE (for Binary serializer only)
- **To have consistent data, after first serialization and storing byte[] into the database - never delete serialized object/class properties.**
- **To have consistent data, after first serialization and storing byte[] into the database - add new properties only to the end of the object/class, after all other properties are listed.**

JSON serialization:

```
var jsonSettings = new Biser.JsonSettings { DateFormat =
Biser.JsonSettings.DateTimeStyle.ISO };
string prettifiedJsonString = new Biser.JsonEncoder(t6,
jsonSet).GetJSON(Biser.JsonSettings.JsonStringStyle.Prettify);
var restoredJsonObject= TS6.BiserJsonDecode(prettifiedJsonString,
settings: jsonSettings);
```

NOTE (for JSON serializer only)
- **JSON serializer can also store multi-dimensional arrays like [,,] [,] [,,,] etc., representing it as a Tuple<List<int>, object itself> where Item1 represents array dimensions.**

Example of the TS6 object for serialization and generated by BiserObjectify Binary and JSON serializer

[20210104] Embedding Biser as a custom serializer for DBreeze

First of all, create necessary partial classes with NuGet package BiserObjectify,
then add static ConcurrentDictionary near DBreezeEngine and such code stubs:

```
using System;


using DBreeze;
using DBreeze.Utils;
using System.Reflection;
using System.Collections.Concurrent;
```

```
static DBreezeEngine DB = null;
static ConcurrentDictionary<string, MethodInfo> BiserTypes = new
ConcurrentDictionary<string, MethodInfo>();

        void Init()
        {
            DB = new DBreezeEngine(@"...your path to DBreeze
folder...");

            CustomSerializator.ByteArraySerializator = (object o) => {

                return ((Biser.IEncoder)o).BiserEncoder().Encode();
            };

            CustomSerializator.ByteArrayDeSerializator = (byte[] bt,
Type t) =>
            {
                var minfo = BiserTypes.GetOrAdd<Type>(t.FullName,(string
typeFullName,Type typeSelf) => {
                    return typeSelf.GetMethod("BiserDecode",
BindingFlags.Public | BindingFlags.Static);
                }, t);

                return minfo.Invoke(null, new object[] { bt, null });
            };
        }
```

[Example Project](#)

[20211004] DBreeze.Utils.MultiKeyDictionary and MultiKeySortedDictionary received generic constructor for .NET 4.7.2>, .NET Standard 2.1> and .NETCore 2>

To search Dictionaries by multiple keys it is possible:

```
Dictionary<(int was, (decimal, decimal)), int> fd6 = new
Dictionary<(int, (decimal, decimal)), int>();
        fd6.Add((1, (12,12)), 1);
        fd6.Add((1, (12, 14)), 2);

        fd6.TryGetValue((1, (12, 14)), out var tzz61);
```

To enhance its functionality with RemoveFromKey and SelectFromKey MultiKeyDictionary (MultiKeySortedDictionary) was created:

```
MultiKeyDictionary.ByteArraySerializator =
ProtobufHelper.SerializeProtobuf;
MultiKeyDictionary.ByteArrayDeSerializator =
ProtobufHelper.DeserializeProtobuf;
```

```
//ProtobufHelper examples can be found here
https://github.com/hhblaze/DBreeze/blob/master/VisualTester/ProtobufHelper.cs
```

```
MultiKeyDictionary<(int was, (int, int), decimal), string> hzu = new
MultiKeyDictionary<(int, (int, int), decimal), string>();
          hzu.Add((1, (2, 2),65m), "dfs1");
          hzu.Add((1, (2, 3),65m), "dfs2");
          hzu.Add((1, (2, 4),65m), "dfs3");
          hzu.Add((2, (2, 2),65m), "dfs4");
          hzu.Add((2, (2, 3),65m), "dfs5");
          hzu.Add((2, (2, 4),65m), "dfs6");
          hzu.Add((3, (2, 2),65m), "dfs7");
          hzu.Add((3, (2, 3),65m), "dfs8");
          hzu.Add((3, (2, 4),65m), "dfs9");

          var tr243254 = hzu.Serialize();

          MultiKeyDictionary<(int was, (int, int), decimal), string>
hzu1 = new MultiKeyDictionary<(int, (int, int), decimal), string>();
          hzu1.Deserialize(tr243254);

          hzu1.Remove(2);

          foreach (var el in hzu1.GetByKeyStart(3))
          {
              Console.WriteLine(el.Item1.was + "__" + el.Item2);
          }
```

Output:

3__dfs7
3__dfs8
3__dfs9

Available operations: *Count*, *Clear, Get, Remove, GetAll, GetByKeyStart, TryGetValue*.

It is possible to **supply keys** in the format of the **ValueTuple** (currently 16 keys are supported, ask / add & recompile for a higher number of keys).

Also, thanks to the beautiful project of Frantisek Konopecky, it is possible to make very **fast DeepCopy** (**Clone**) of **any object** (faster than protobuf's serialize-deserialize about 3 times) and, of course, it is possible **to clone** the **MKD**:

    var newmkd = hzu.**CloneMultiKeyDictionary**();

    It is integrated into DBreeze, use it with any object like:

```
using DBreeze.Utils;

MyObject Clone()
{
    return (new MyObject()).CloneByExpressionTree();
}
```

The code limitations for the DeepCopy :
- **Does not copy delegates and events (leaves null instead)**
- **Fails on ComObjects (e.g. on some WPF dispatcher subobjects or Excel Interop)**
- **Fails on any unmanaged object (e.g. from some external C++ library)**

MKD obeys the same **synchronization** rules as Dictionary, so access from different threads must be via lock:

```
        ReaderWriterLockSlim _sync = new ReaderWriterLockSlim();
        MultiKeyDictionary<(int cid, int wid, float aid), string> mkd41
 = new MultiKeyDictionary<(int, int, float), string>();


        private void ReadFrom_MKD((int cid, int wid, float aid) newKey,
 string newValue)
        {
            _sync.EnterReadLock();
            try
            {
                mkd41.Add(newKey, newValue);

            }
```

```csharp
            finally
            {
                _sync.ExitReadLock();
            }
        }


        private void WriteInto_MKD((int cid, int wid, float aid) newKey,
string newValue)
        {
            _sync.EnterWriteLock();
            try
            {
                mkd41.Add(newKey, newValue);

            }
            finally
            {
                _sync.ExitWriteLock();
            }
        }


        private void WriteInto_MKD_IfKeyNotFound((int cid, int wid,
float aid) newKey, string newValue)
        {
            _sync.EnterUpgradeableReadLock();
            try
            {
                if (!mkd41.Contains(newKey)) //emulating write ONLY in
case when key was not found
                {
                    _sync.EnterWriteLock();
                    try
                    {
                        if (!mkd41.Contains(newKey)) //recheck
                        {
                            mkd41.Add(newKey, newValue);
                        }

                    }
                    finally
                    {
                        _sync.ExitWriteLock();
                    }
```

```
                }
            }
            finally
            {
                _sync.ExitUpgradeableReadLock();
            }
        }
```

[20211005] **await inside DBreezeEngine transaction**

It is recommended to avoid await inside transactions, especially for long running operations
and move it out of DBreezeEngine.Transaction.
But if it is really necessary, it must be decorated with Wait() to preserve context return to the
same ManagedThreadId in case when after await should come insert (tran.Insert) block:

```
using(var tran in engine.GetTransaction())
{

tran.Select
tran.Insert

Var result = (await HttpHandler.Get("url")).Wait();

tran.Insert

tran.Commit
}
```

DBreeze can read data inside one transaction from the parallel threads, but has integrated
defenses from the tables being written within the same transaction from different
ManagedThreads. Parallel transactions will be queued, but the same transaction, trying to
write from different threads, will be thrown out with the exception.

[20211208] **overloads for SelectBackwardFromTo and SelectForwardFromTo**

It is coming from the request:

"we often have to read time based records, where the key is the date (start date of event). For example: there are events at 8:00, 12:00, 16:00 (all events are not overlapping) when we request all events since 10:00-14:00, we have to do selectForwardFromTo(10:00, 14:00) + selectBackward(fromTo(10:00, minValue).Take(1), because we always have to include also "minus 1" record in past to check - maybe this event was running till 10:00.
"

To the mentioned functions were added overloads with the parameter **int grabSomeLeadingRecords.** Now we can do something like this:

```
using (var t = dbe.GetTransaction())
            {
                for (int i = 0; i < 10; i++)
                    t.Insert<int, int>("t1", i, i);

t.Commit();

            }


  using (var t = dbe.GetTransaction())
            {
                foreach(var row in
 t.SelectBackwardFromTo<int,int>("t1",7,true,3,true,2))
                {
                    Console.WriteLine(row.Key);

                }
                Console.WriteLine("-------");
                foreach (var row in t.SelectForwardFromTo<int,
 int>("t1", 5, true, 9, true, 2))
                {
                    Console.WriteLine(row.Key);

                }
            }
```

If **grabSomeLeadingRecords** tries to grab non-existing values - it is not a problem, non-existing values will be skipped.

Result:
9
8
7
6
5
4

```
3
-------
3
4
5
6
7
8
9
```

[20211213] Reading of the same structured keys from different tables simultaneously. Multi_SelectForwardFromTo && Multi_SelectBackwardFromTo

It is coming from the idea that we have in different tables the same structured data, but stored from different sources, e.g. events for the Location1 (Hamburg) are stored in the table "EventsLocation1" and events for the Location2 (Berlin) are stored in the table "EventsLocation2". Keys are datetimes, Values are a kind of EventInLocation object.
Now we want to read all events sorted by key from both locations.

More about the request can be read from here

For that - 2 new functions were added: Multi_SelectForwardFromTo and Multi_SelectBackwardFromTo.

Example of the insert:

```csharp
using (var tran = dbe.GetTransaction())
        {
            tran.SynchronizeTables("t*");

            for (int i = 0; i < 20; i++)
            {
                tran.Insert<int, int>("t1", i, i);
                tran.Insert<int, int>("t2", ++i, i);
                tran.Insert<int, int>("t3", ++i, i);
            }
            // Manual values for duplicating keys between tables
            tran.Insert<int, int>("t1", 5, 50);
            tran.Insert<int, int>("t2", 8, 80);
            tran.Insert<int, int>("t3", 15, 150);
            tran.Commit();
        }
```

Here is an example of getting in sorted order all elements from 4 different tables with the

same key structure. **Note**, that table "t4" doesn't exist at all:

```csharp
using (var tran = dbe.GetTransaction()){
  HashSet<string> tables = new HashSet<string>() { "t1", "t4", "t2", "t3"
};

                foreach (var el in tran.Multi_SelectForwardFromTo<int,
 int>(tables, int.MinValue, true, int.MaxValue, true))
                {
                    Console.WriteLine($"tbl: {el.TableName}: Key:
 {el.Key}; Value: {el.Value}");
                }

                Console.WriteLine("-------------------------------");

                foreach (var el in tran.Multi_SelectBackwardFromTo<int,
 int>(tables, int.MaxValue, true, int.MinValue, true))
                {
                    Console.WriteLine($"tbl: {el.TableName}: Key:
 {el.Key}; Value: {el.Value}");
                }

}
```

Result:

tbl: t1: Key: 0; Value: 0
tbl: t2: Key: 1; Value: 1
tbl: t3: Key: 2; Value: 2
tbl: t1: Key: 3; Value: 3
tbl: t2: Key: 4; Value: 4
tbl: t1: Key: 5; Value: 50
tbl: t3: Key: 5; Value: 5
tbl: t1: Key: 6; Value: 6
tbl: t2: Key: 7; Value: 7
tbl: t2: Key: 8; Value: 80
tbl: t3: Key: 8; Value: 8
tbl: t1: Key: 9; Value: 9
tbl: t2: Key: 10; Value: 10
tbl: t3: Key: 11; Value: 11
tbl: t1: Key: 12; Value: 12
tbl: t2: Key: 13; Value: 13
tbl: t3: Key: 14; Value: 14
tbl: t1: Key: 15; Value: 15
tbl: t3: Key: 15; Value: 150
tbl: t2: Key: 16; Value: 16
tbl: t3: Key: 17; Value: 17

tbl: t1: Key: 18; Value: 18
tbl: t2: Key: 19; Value: 19
tbl: t3: Key: 20; Value: 20
----
tbl: t3: Key: 20; Value: 20
tbl: t2: Key: 19; Value: 19
tbl: t1: Key: 18; Value: 18
tbl: t3: Key: 17; Value: 17
tbl: t2: Key: 16; Value: 16
tbl: t1: Key: 15; Value: 15
tbl: t3: Key: 15; Value: 150
tbl: t3: Key: 14; Value: 14
tbl: t2: Key: 13; Value: 13
tbl: t1: Key: 12; Value: 12
tbl: t3: Key: 11; Value: 11
tbl: t2: Key: 10; Value: 10
tbl: t1: Key: 9; Value: 9
tbl: t2: Key: 8; Value: 80
tbl: t3: Key: 8; Value: 8
tbl: t2: Key: 7; Value: 7
tbl: t1: Key: 6; Value: 6
tbl: t1: Key: 5; Value: 50
tbl: t3: Key: 5; Value: 5
tbl: t2: Key: 4; Value: 4
tbl: t1: Key: 3; Value: 3
tbl: t3: Key: 2; Value: 2
tbl: t2: Key: 1; Value: 1
tbl: t1: Key: 0; Value: 0

If the keys structure differs (for the selected key scope ONLY) - exception will be thrown.

[20211227] `Integrated CloneByExpressionTree and CPU time economy on huge amounts of object instantiations.`

We got a real life scenario where we had to initialize one object about 66 MLN. times within 40 days and only this init process took about 4 hours 30 minutes, using standard .NET "=

new MyObject()".

Object itself has about 50 properties and 25 of them have non-default values that are assigned in the constructor. Not all properties are simple (ByVal): 7 of them are instances of other classes (ByRef).

Here will be described the trick with `CloneByExpressionTree` (described earlier in the docu) that helped to decrease time from 4.5 hours up to 7 minutes (40 times).

First we instantiate a prototype object and store it somewhere in the code (usually it can be a static variable visible within the whole library)

```
public static var myObjProto = new MyObject();
```

Then, each time when we need to create a new instance we make following:

```
var instanceOfMyObj1 = myObjProto.CloneByExpressionTree();
var instanceOfMyObj2 = myObjProto.CloneByExpressionTree();
```

instead of `instanceOfMyObj1 = new MyObject();`

[20231103] DBreeze as an Embedding Vector Database / Similarity Search Engine / Clustering, based on KMeans++ Centroids Propagated Graph.

This layer is undergoing intensive testing and moved to beta.
However, it is recommended to use it and to become familiar with the concept.

From now DBreeze has an integrated layer for storing embedding vectors and searching most relevant from them to the query - vector database and vector similarity search engine .

Very big Thanks to those kind guys for the theory:

Yu. A. Malkov, D. A. Yashunin "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs"
https://arxiv.org/ftp/arxiv/papers/1603/1603.09320.pdf

Curiosity-AI and Theolivenbaum
https://github.com/curiosity-ai/hnsw-sharp

*This functionality is available in DBreeze for **.NET Framework 4.72>, .NET6>**
.NetCoreApp3.1> .NetStandard2.1>*

A bit of WIKI:

Embedding vectors are numerical representations of items such as words, sentences, documents, or even images, in a continuous vector space. These vectors are often created using techniques like word embeddings (e.g., Word2Vec, GloVe) for text data or image embeddings (e.g., CNN-based embeddings) for image data. The main purpose of embedding vectors in the context of a database can include:

1. Semantic Analysis:
Embedding vectors help in capturing semantic relationships between items. For example, in natural language processing, similar words are mapped to nearby points in the embedding space. This semantic understanding can be valuable in applications like search, recommendation systems, and sentiment analysis.

2. Efficient Similarity Search:
Embedding vectors enable efficient similarity search in high-dimensional spaces. Traditional distance metrics like cosine similarity or Euclidean distance can be used to measure similarity between vectors. This is particularly useful in applications where finding similar items quickly, such as in recommendation systems or image retrieval, is important.

3. Recommendation Systems:
Embedding vectors can represent users and items in recommendation systems. By learning embeddings for users and items, recommendation algorithms can efficiently find similar items for a user. This is fundamental to providing personalized content or product recommendations on platforms like e-commerce websites or streaming services.

4. Information Retrieval:
Embedding vectors can enhance information retrieval systems. By representing documents or queries as vectors, it's easier to find relevant documents quickly, especially in large datasets. This is crucial in search engines, where users expect fast and accurate results.

5. Clustering and Classification:
Embedding vectors can be used as features for clustering or classification tasks. Machine learning algorithms can work with these vectors to group similar items together (clustering) or categorize items into predefined classes (classification).

6. Anomaly Detection:
Anomalies or outliers in datasets often do not conform to the patterns found in regular data points. Embedding vectors can be used to detect such anomalies by identifying data points that are far away from the dense regions in the embedding space.

7. Language Translation:
In machine translation tasks, words or sentences from one language are embedded into vectors and then translated by transforming these vectors into another language's embedding space. This method often yields better translation results compared to traditional statistical machine translation methods.

In summary, embedding vectors provide a powerful way to represent complex data in a simplified, numerical form, enabling efficient processing, analysis, and understanding of the underlying patterns in various applications.

END OF WIKI

In two steps, we generate a multidimensional array from any object using a neural network. Next, we create a multidimensional array from the search query. We then determine which object is closer to our query. When creating the object's feature representation array, the neural network considers various parameters of the object, including semantic relationships (in the case of text).

There are some other Embedding Vector Databases on the market, like https://qdrant.tech/ etc. Read more about embedding vectors on their website or OpenAI website or https://towardsdatascience.com/introduction-to-embedding-clustering-and-similarity-11dd80b00061.

To obtain a vector representation of a text, image, or any other data, a specialized neural network is required. These networks translate objects into arrays of doubles[] or floats[] with varying dimensionalities, such as 100 or 300. For instance, OpenAI's text-embedding-ada-002 generates vectors consisting of 1536 double elements.

Example of global external services to get embedding vectors from your data:
Multimodal embeddings Google
Text embeddings Google
OpenAI embeddings - OpenAI access layer reflected here (you need OpenAI APIKey)
Custom Image Similarity etc.

Also it is possible to do locally e.g with ML.NET, using models that ML.NET will automatically download for you on first usage (note, it can take some time and your program will be unresponsive during the first load, depending on the model size, located in AppData\Local\mlnet-resources\WordVectors):
https://learn.microsoft.com/en-us/dotnet/api/microsoft.ml.textcatalog.applywordembedding?view=ml-dotnet
It is reflected in GetSomeEmbeddingVectors()

Or call transformers from Python

How to use DBreeze Vector layer:

You will need an extra DBreeze table to store vectors. Each embedding vector must represent a kind of a document (text, image etc.) or many document chunks. Documents must be stored in one table, embedding vectors in another table. Vector tables should not be touched by the programmer directly, they are served internally by DBreeze.
Of course, different types of documents, for different purposes, may be stored in different

"Vector" tables.

The dimensionality of vectors in one table must be the same.

The rest is easy and examples are represented in
https://github.com/hhblaze/DBreeze/blob/master/TesterNet6/TextCorpus/ITGiantLogotypes.cs

First, we store vectors obtained from OpenAI (or the other Neural Network) inside function Store_Docs_Vectors(), then finding most relevant with SearchLogo();

All concerning the Vector Layer inside DBreeze transaction starts from **tran.Vector**…

It is possible to **Add** DocumetID+Vectos, **Remove** DocumentID and finally to **SearchSimilar**.

Vector tables are the common DBreeze tables, that's why we synchronize ***them, and*** commit ***in the end, when inserting.***

While inserting deferred indexing is possible (Note, after upgrading vector layer - not necessary).

Here is an example of inserting and searching taken from
https://github.com/hhblaze/DBreeze/blob/master/TesterNet6

In this example we have created from each element of
https://github.com/hhblaze/DBreeze/blob/master/TesterNet6/TextCorpus/ITGiantLogotypes.json a document, vectorize each of them, put into DB; then we search which document corresponds better (more similar, has closer distance in multidimensional space) to our question, like "`rectangle like logo`" or "`fruit like logo`".

Note: the origin of embedding vectors for the documents and embedding vectors for the questions must be the same, actually the same pre-trained Neural Network.

```
public static class ITGiantLogotypes
  {
      static string tblKNNITLogos = "KNNITLogos"; //Vector Table for ITLogos
      static string tblDocsITLogos = "DocsITLogos"; //Docs ItLogos
```

```csharp
        public async static Task Example()
        {
            await ITGiantLogotypes.Store_Docs_Vectors();
            await ITGiantLogotypes.SearchLogo();
        }

        public class DBLogotype
        {
            public ITGiantLogotypesJson Company { get; set; }
            public double[] Embedding { get; set; }
        }

        public class ITGiantLogotypesJson
        {
            public string Company { get; set; }
            public string LogoDescription { get; set; }

        }

        public async static Task SearchLogo()
        {

            //Those for tests....
            string question = "fruit like logo";
            //question = "square like logo";
            question = "rectangle like logo";
            question = "multicolor logo";

            //-getting embedding vector for the question
            var emb = await OpenAI.GetEmbedding(question);


            if (emb == null && emb.error)
                throw new Exception("Can't get embedding from the question");

            //-bringing to array
            double[] questionEmbedding = emb.EmbeddingAnswer.ToArray();

            //-show top 3 most relevant answers
            using (var tran = Program.DBEngine.GetTransaction())
            {
                tran.ValuesLazyLoadingIsOn = false; //to read key already with value

                var res = tran.VectorsSearchSimilar(tblKNNITLogos, questionEmbedding,
3);

                foreach (var el in res)
                {
                    var rowDoc = tran.Select<byte[], string>(tblDocsITLogos,
2.ToIndex(el.ExternalId));
                    var dbCompany =
JsonSerializer.Deserialize<DBLogotype>(rowDoc.Value);
                    Console.WriteLine($"Company: {dbCompany.Company.Company}");
```

```csharp
                Console.WriteLine($"\tDescription:
{dbCompany.Company.LogoDescription}");

            }
        }

    }


        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public async static Task Store_Docs_Vectors()
        {

            //-to skip this run again, checking if data already exists
            using (var tran = Program.DBEngine.GetTransaction())
            {
                int idCnt = tran.Select<byte[], int>(tblDocsITLogos, 1.ToIndex()).Value;
                if (idCnt > 0)
                    return;
            }


            /*
                -Reading data from json,
                - Adding each element as a document
                - Store embedding vectors of each document in a vector table (has own
name) for this document type
             */
            DBLogotype slt = null;

            var itg =
JsonSerializer.Deserialize<List<ITGiantLogotypesJson>>(File.ReadAllText(@"..\..\..\TextC
orpus\ITGiantLogotypes.json"));

            List<DBLogotype> embeddings = new List<DBLogotype>();

            foreach (var el in itg)
            {
                //-Getting vector of the text: el.Company + " " + el.LogoDescription
                var emb = await OpenAI.GetEmbedding(el.Company + " " +
el.LogoDescription);

                if (emb != null && !emb.error)
                {
                    slt = new DBLogotype
                    {
                        Company = el,
                        Embedding = emb.EmbeddingAnswer.ToArray(), //converting to
double[]
                    };
```

```csharp
                    embeddings.Add(slt);
                }
            }


            if (embeddings.Count > 0)
            {

                using (var tran = Program.DBEngine.GetTransaction())
                {
                    //-sync of Doctable and vector table for searching docs
                    tran.SynchronizeTables(tblKNNITLogos, tblDocsITLogos);

                    //Creating documents of it
                    int idCnt = tran.Select<byte[], int>(tblDocsITLogos,
1.ToIndex()).Value;

                    //-such format will be inserted into VectorTable, Key is exernal
documentID, value is vector itself
                    Dictionary<byte[], double[]> vectorsToInsert = new
Dictionary<byte[], double[]>();

                    foreach (var el in embeddings)
                    {
                        idCnt++;
                        //-storing doc itself (not necessary to store embedding vector,
it will be stored in vector table, but we do it here for tests, to skip next time
acquiring from OpenAI)
                        tran.Insert<byte[], string>(tblDocsITLogos, 2.ToIndex(idCnt),
JsonSerializer.Serialize(el));

                        //accumulating vectors in Dictionary (bringing toArray
el.Value.Item2.ToArray())
                        vectorsToInsert.Add(idCnt.To_4_bytes_array_BigEndian(),
el.Embedding);
                    }

                    //-storing Doc Index monotonically growing
                    if (embeddings.Count > 0)
                    {
                        tran.Insert<byte[], int>(tblDocsITLogos, 1.ToIndex(), idCnt);
                    }

                    //-storing documents as vectors (with/without deferred indexing)
                    if (vectorsToInsert.Count > 0)
                    {
                        //-in case of big quantity of vectors, use deferredIndexing:
true (to run computation in the background)
                        tran.VectorsInsert(tblKNNITLogos, vectorsToInsert,
deferredIndexing: false);
                    }

                    tran.Commit();
                }
```

```
                }


            }//eof



        }//eoc
```

Clustering KMeans++

https://en.wikipedia.org/wiki/K-means_clustering

If you have a table with vectors you can try to split it into N several clusters and check which vectors belong to which cluster. Second option you can create N clusters from your documents and see which others documents will be assigned to your N clusters.

Use tran.**VectorsClusteringKMeans**

Example can be found here: it is based on the previous data about ITGiantLogos

```csharp
 internal class Clustering
    {
        static string tblKNNITLogos = "KNNITLogos"; //Vector Table for ITLogos
        static string tblDocsITLogos = "DocsITLogos"; //Docs ItLogos

        public static void KMeansTest()
        {

            //----------FIRST APPROACH

            //-from all items in tblKNNITLogos we try to create 2 random clusters
            using (var tran = Program.DBEngine.GetTransaction())
            {
                tran.ValuesLazyLoadingIsOn = false; //to read key already with value

                var res = tran.VectorsClusteringKMeans(tblKNNITLogos, 2);

                foreach (var el in res)
                {
                    foreach (var doc in el.Value)
                    {
                        Console.WriteLine($"CLUSTER: {el.Key}");
```

```csharp
                        var rowDoc = tran.Select<byte[], string>(tblDocsITLogos,
2.ToIndex(doc));
                        var dbCompany =
JsonSerializer.Deserialize<DBLogotype>(rowDoc.Value);
                        Console.WriteLine($"Company: {dbCompany.Logotype.Company}");
                        Console.WriteLine($"\tDescription:
{dbCompany.Logotype.LogoDescription}");

                }

            }
        }//eo using


        //----------SECOND APPROACH


        //-from all items in tblKNNITLogos we try to create clusters around
specified documents (taken their vectors as prototypes for the clusters)
        using (var tran = Program.DBEngine.GetTransaction())
        {
            tran.ValuesLazyLoadingIsOn = false; //to read key already with value

            List<byte[]> twoClusters=new List<byte[]>();
            foreach(var row in tran.SelectForwardStartsWith<byte[],
string>(tblDocsITLogos, 2.ToIndex()))
            {
                var dbCompany = JsonSerializer.Deserialize<DBLogotype>(row.Value);
                //Console.WriteLine($"{row.Key.Substring(1).To_Int32_BigEndian()}
Company: {dbCompany.Logotype.Company}");

                if (dbCompany.Logotype.Company == "MICROSOFT")
                    twoClusters.Add(row.Key.Substring(1));
                if (dbCompany.Logotype.Company == "META")
                    twoClusters.Add(row.Key.Substring(1));
            }

            var res = tran.VectorsClusteringKMeans(tblKNNITLogos, 0,
externalDocumentIDsAsCentroids: twoClusters);

            foreach (var el in res)
            {
                foreach (var doc in el.Value)
                {
                    Console.WriteLine($"CLUSTER: {el.Key}");
                    var rowDoc = tran.Select<byte[], string>(tblDocsITLogos,
2.ToIndex(doc));
                    var dbCompany =
JsonSerializer.Deserialize<DBLogotype>(rowDoc.Value);
                    Console.WriteLine($"Company: {dbCompany.Logotype.Company}");
                    Console.WriteLine($"\tDescription:
{dbCompany.Logotype.LogoDescription}");
```

```
                }

            }
        }//eo using


    }//eof



}//eoc
```

Determining to which cluster new documents belong

For example we have already 10 different clusters with plenty of documents and we want to quickly determine to which of those clusters belong new incoming documents. We can take from each cluster a really representative document (its embedding vector) and define that this vector is a cluster prototype. In this case we will have 10 prototype vectors. The only thing now we should do is to use the tran.VectorsClusteringKMeans variation after getting embedding-vectors of all new documents.

Note: it is possible to get vector from existing vector table by tran.**VectorsGetByExternalDocumentIDs**<double[]>

This solution is described in function KMeansFindCluster from
https://github.com/hhblaze/DBreeze/blob/master/TesterNet6/TextCorpus/Clustering.cs


```csharp
  public static async Task KMeansFindCluster()
  {
      //-getting embeddings for each item of @"..\..\..\TextCorpus\FurnitureV1.json",
further we will work only with @"..\..\..\TextCorpus\FurnitureV1withEmbeddings.json"
      //await GetFunrnitureV1Embeddings();

      var furnitureLst =
JsonSerializer.Deserialize<List<FurnitureV1>>(File.ReadAllText(@"..\..\..\TextCorpus\Fur
nitureV1withEmbeddings.json"));

      using (var tran = Program.DBEngine.GetTransaction())
      {
          //Flattering cluster prototypes and itemsTobeClustered
          Dictionary<int, (string ClusterName, FurnitureItem Furniture)>
clusterPrototypes = new Dictionary<int, (string clusterName, FurnitureItem)>();
```

```csharp
        Dictionary<int, (string ClusterName, FurnitureItem Furniture)>
itemsToBeClustered = new Dictionary<int, (string clusterName, FurnitureItem)>();

        //Let's take first element from each Cluster as a Cluster prototype and the
rest of elements will have to find to which cluster they belong to
        int i = 0;
        int j = 0;
        foreach (var cluster in furnitureLst)
        {
            var first = cluster.Items.First();
            clusterPrototypes.Add(i,(cluster.Cluster,first));

            foreach (var item in cluster.Items.Skip(1)) //skipping first value from the
cluster and adding to itemsToBeClustered
            {
                itemsToBeClustered.Add(j, (cluster.Cluster, item));
                j++;
            }
            i++;
        }


        //-supplying two Lists of embeddings, first are prototypes that represent
different clusters and second are items to be sparsed between all prototypes clusters
        var res = tran.VectorsClusteringKMeans(
            clusterPrototypes.Values.Select(r=>r.Item2.Embedding).ToList(),
            itemsToBeClustered.Values.Select(r => r.Item2.Embedding).ToList()
            );

        //resulting output
        i = 0;
        j = 0;
        foreach (var el in res)
        {
            Console.WriteLine($"CLUSTER:
-------------{clusterPrototypes[i].ClusterName}-------------");
            foreach (var item in el.Value)
            {
                Console.WriteLine($"\t OriginalCluster:
{itemsToBeClustered[item].ClusterName}; Name:
{itemsToBeClustered[item].Furniture.Name}");
                j++;
            }
            i++;
        }

    }


  }
```

We got such data corpus and its modified version already with embedding vectors for each item FurnitureV1withEmbeddings.json. We take the first item of each cluster as a prototype

element for the cluster and put all other elements as unknown items. We want to find out which clusters will be bound by KMeans.

Here is the result:

CLUSTER: ------------Sofa------------
    OriginalCluster: Sofa; Name: Rustic Wooden Sofa
    OriginalCluster: Sofa; Name: Chic Linen Settee
    OriginalCluster: Sofa; Name: Family-Friendly Sofa Bed
    OriginalCluster: Sofa; Name: Cozy Corner Sectional
    OriginalCluster: Sofa; Name: Modern Elegance Couch
    OriginalCluster: Sofa; Name: Luxury Leather Loveseat
    OriginalCluster: Sofa; Name: Minimalist Upholstered Couch
    OriginalCluster: Sofa; Name: Contemporary Reclining Sofa
    OriginalCluster: Sofa; Name: Vintage Velvet Settee
CLUSTER: ------------Chairs------------
    OriginalCluster: Chairs; Name: Sleek Metal Bar Stool
    OriginalCluster: Chairs; Name: Velvet Accent Chair
    OriginalCluster: Chairs; Name: Foldable Butterfly Chair
    OriginalCluster: Chairs; Name: Classic Wooden Dining Chair
    OriginalCluster: Chairs; Name: Mid-Century Modern Armchair
    OriginalCluster: Chairs; Name: Plush Lounge Chair
    OriginalCluster: Chairs; Name: Antique Rocking Chair
    OriginalCluster: Chairs; Name: Swivel Rocking Chair
    OriginalCluster: Kids' Furniture; Name: Kid-Sized Rocking Chair
CLUSTER: ------------Tables------------
    OriginalCluster: Tables; Name: Industrial Style End Table
    OriginalCluster: Tables; Name: Vintage Bedside Table
    OriginalCluster: Tables; Name: Foldable Card Table
    OriginalCluster: Tables; Name: Elegant Marble Console Table
    OriginalCluster: Tables; Name: Versatile Folding Table
    OriginalCluster: Tables; Name: Modern Glass Coffee Table
    OriginalCluster: Tables; Name: Rustic Wooden Side Table
    OriginalCluster: Tables; Name: Adjustable Height Standing Desk
CLUSTER: ------------Beds------------
    OriginalCluster: Beds; Name: Mid-Century Modern Bed
    OriginalCluster: Beds; Name: Trundle Bed for Guests
    OriginalCluster: Beds; Name: Canopy Bed with Sheer Curtains
    OriginalCluster: Beds; Name: Rustic Wooden Bed Frame
    OriginalCluster: Beds; Name: Tufted Velvet Daybed
    OriginalCluster: Beds; Name: Four-Poster Bed with Carvings
    OriginalCluster: Beds; Name: Minimalist Platform Bed
    OriginalCluster: Beds; Name: Adjustable Massage Bed
CLUSTER: ------------Storage------------
    OriginalCluster: Decor; Name: Mirrors with Decorative Frames
    OriginalCluster: Storage; Name: Bathroom Vanity with Storage
    OriginalCluster: Storage; Name: Bookcase with Adjustable Shelves
    OriginalCluster: Storage; Name: Modular Closet System
    OriginalCluster: Storage; Name: Floating Wall Shelves
    OriginalCluster: Decor; Name: Hanging Wall Shelves for Displays
    OriginalCluster: Storage; Name: Storage Ottoman with Tray
    OriginalCluster: Storage; Name: Rattan Basket Storage Unit
    OriginalCluster: Storage; Name: Drawer Chest with Geometric Patterns
    OriginalCluster: Storage; Name: Entryway Hall Tree
CLUSTER: ------------Lighting------------

OriginalCluster: Lighting; Name: Table Lamp with Ceramic Base
OriginalCluster: Lighting; Name: Ceiling Fan with Remote Control
OriginalCluster: Lighting; Name: LED Strip Lights for Ambient Lighting
OriginalCluster: Decor; Name: Candle Holders with Metallic Finish
OriginalCluster: Lighting; Name: Smart Home Lighting System
OriginalCluster: Lighting; Name: Task Lighting Desk Lamp
OriginalCluster: Lighting; Name: Floor Lamp with Adjustable Arm
OriginalCluster: Lighting; Name: Wall Sconces with Vintage Appeal
OriginalCluster: Lighting; Name: Pendant Lights with Geometric Shades
CLUSTER: -------------Decor-------------
OriginalCluster: Decor; Name: Artificial Plants for Indoor Greenery
OriginalCluster: Decor; Name: Handwoven Area Rug with Boho Vibes
OriginalCluster: Decor; Name: Ethnic Throw Blankets
OriginalCluster: Decor; Name: Handcrafted Ceramic Vases
OriginalCluster: Decor; Name: Wall Art Prints with Inspirational Quotes
OriginalCluster: Decor; Name: Framed Family Photos and Memories
CLUSTER: -------------Outdoor Furniture-------------
OriginalCluster: Outdoor Furniture; Name: Hammock with Wooden Stand
OriginalCluster: Outdoor Furniture; Name: Sun Loungers with Umbrella
OriginalCluster: Outdoor Furniture; Name: Outdoor Sectional Sofa Set
OriginalCluster: Outdoor Furniture; Name: Adirondack Chairs with Side Table
OriginalCluster: Outdoor Furniture; Name: Swing Chair with Canopy
OriginalCluster: Outdoor Furniture; Name: Garden Bench with Floral Motifs
OriginalCluster: Outdoor Furniture; Name: Fire Pit Table with Chairs
OriginalCluster: Outdoor Furniture; Name: Foldable Picnic Table and Bench Set
OriginalCluster: Chairs; Name: Outdoor Patio Chair
OriginalCluster: Outdoor Furniture; Name: Outdoor Bar Cart with Wheels
OriginalCluster: Lighting; Name: Outdoor Solar Lanterns
OriginalCluster: Tables; Name: Outdoor Picnic Table
CLUSTER: -------------Kids' Furniture-------------
OriginalCluster: Beds; Name: Bunk Bed with Storage Drawers
OriginalCluster: Kids' Furniture; Name: Space-Themed Bunk Bed with Slide
OriginalCluster: Kids' Furniture; Name: Treehouse-Inspired Bunk Bed
OriginalCluster: Kids' Furniture; Name: Interactive Learning Table
OriginalCluster: Kids' Furniture; Name: Car-Shaped Bed for Kids
OriginalCluster: Kids' Furniture; Name: Princess-Themed Vanity Set
OriginalCluster: Kids' Furniture; Name: Toy Storage Chest with Safety Hinges
OriginalCluster: Kids' Furniture; Name: Animal-Themed Desk and Chair Set
OriginalCluster: Kids' Furniture; Name: Educational Alphabet Rug
OriginalCluster: Storage; Name: Cubby Organizer for Kids
CLUSTER: -------------Pet Furniture-------------
OriginalCluster: Pet Furniture; Name: Self-Cleaning Litter Box
OriginalCluster: Pet Furniture; Name: Scratch-Resistant Cat Tree
OriginalCluster: Pet Furniture; Name: Elevated Dog Bed with Canopy
OriginalCluster: Pet Furniture; Name: Pet Feeding Station with Storage
OriginalCluster: Pet Furniture; Name: Orthopedic Dog Sofa Bed
OriginalCluster: Pet Furniture; Name: Pet Grooming Station with Tools
OriginalCluster: Pet Furniture; Name: Collapsible Pet Playpen
OriginalCluster: Pet Furniture; Name: Interactive Treat-Dispensing Toy
OriginalCluster: Pet Furniture; Name: Cat Hammock Window Perch

This result is quite good, having the fact the vectors were built up from very small text

information about this furniture.

Benchmarking VectorsInsert.

Each time by adding a new vector to the table, the Vector Layer must smartly recompute connections between graph nodes, where each embedding vector is a node, to guarantee the fast embedding search.

Each vector can represent an image or a pdf document or document chunk (depending upon the payload of the embedding vectors Neural Network, PDF can be of 20-50-... pages).

On a standard PC we can talk about such digits:

```csharp
public static void LoadV1()
{
    string tblLoadV1 = "LoadV1";
    DBreeze.Utils.FastRandom frnd=new DBreeze.Utils.FastRandom();

    int batchSize = 1000; //1000 (batchSize) documents per round
    double[][] bt = new double[batchSize][];

    System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
    System.Diagnostics.Stopwatch sw1 = new System.Diagnostics.Stopwatch();
    sw.Start();
    DateTime now = DateTime.Now;
    for (int j = 0; j < 100; j++)//100 times we insert a batch
    {
        Console.Write($"{(j * batchSize + batchSize)} - "); //quantity of
documents inside

        for (int i = 0; i < batchSize; i++)//inserting vectors of size 500
double
        {

            bt[i] = new double[500];
            for(int pp = 0;  pp<500; pp++)
                bt[i][pp] = frnd.NextDouble();

        }
        sw1.Reset();
        sw1.Start();
        using (var tran = Program.DBEngine.GetTransaction())
        {
            var x = bt.Select((k, v) =>
            new KeyValuePair<byte[], double[]>(
                (v + j * batchSize).ToBytes(),
                k.Select(Convert.ToDouble).ToArray()))
```

```
                .ToDictionary(k => k.Key, v => v.Value);


            tran.VectorsInsert(tblLoadV1, x);

            tran.Commit();
        }
        sw1.Stop();
        Console.WriteLine($"roundMS: {sw1.ElapsedMilliseconds}");
    }
    sw.Stop();
    Console.WriteLine($"MS: {sw.ElapsedMilliseconds}; - {(DateTime.Now -
 now).TotalMilliseconds}");
 }
```

We insert 100 batches with 1000 documents (each document represented with the vector double[500]).

```
57000 - roundMS: 370
58000 - roundMS: 367
59000 - roundMS: 284
60000 - roundMS: 249
61000 - roundMS: 267
62000 - roundMS: 307
63000 - roundMS: 380
64000 - roundMS: 333
65000 - roundMS: 472
66000 - roundMS: 394
67000 - roundMS: 386
68000 - roundMS: 350
69000 - roundMS: 267
70000 - roundMS: 337
71000 - roundMS: 243
72000 - roundMS: 375
73000 - roundMS: 472
74000 - roundMS: 504
75000 - roundMS: 338
76000 - roundMS: 250
77000 - roundMS: 297
78000 - roundMS: 364
79000 - roundMS: 370
80000 - roundMS: 395
81000 - roundMS: 291
82000 - roundMS: 267
83000 - roundMS: 266
84000 - roundMS: 458
85000 - roundMS: 254
86000 - roundMS: 224
87000 - roundMS: 332
88000 - roundMS: 397
89000 - roundMS: 327
90000 - roundMS: 389
91000 - roundMS: 379
92000 - roundMS: 340
93000 - roundMS: 421
94000 - roundMS: 406
95000 - roundMS: 276
96000 - roundMS: 298
97000 - roundMS: 490
98000 - roundMS: 242
99000 - roundMS: 401
100000 - roundMS: 297
MS: 35391; - 35391,1288
Press any key
```

Insert of 100batch/1000docs each, double [500] vectors, already indexed for similarity search

1000*100*500*8(bytes) it should reside around 400MB, but the table takes only 269MB on disk due to GZIP compression of graph nodes containing vectors. RAM and CPU usage is very low. Similarity search speed is blazing fast.

Because embedding **vector tables** are common DBreeze tables, they **can be located in memory**, when necessary. It is possible to write from different threads in parallel to different vector tables.

Vector Database on Mobile Devices optimizations

Starting from Release 1.117 there is a new Dbreeze configuration is possible for the Vector layer

```
DBreezeConfiguration conf = new DBreezeConfiguration()
{
    DBreezeDataFolderName = DBPath,
    Storage = DBreezeConfiguration.eStorage.DISK,
    VectorLayerConfig = new DBreezeConfiguration.VectorlayerConfiguration()
    {
        Dense = 1000 //For the mobile devices value can be set to 100, 1000 is the default value
    }
};
```


If something is not working like it is expected, please, don't hesitate to write down an issue comment on https://github.com/hhblaze/DBreeze or http://dbreeze.tiesky.com

Copyright © 2012 dbreeze.tiesky.com / Oleksiy Solovyov / Ivars Sudmalis