DBreeze Database Documentation.

(DBreeze version 01.059.20130812; doc. version 01.029.20130812)

Professional, open-source, NoSql (embedded Key/Value storage), transactional, ACID-compliant, multi-threaded database management system for C# .NET 3.5> MONO.

Alex Solovyov < hhblaze@gmail.com >
Ivars Sudmalis < zikills@gmail.com >

It's a free software for those, who thinks that it should be free.

Please, notify us about our software usage, so we can evaluate and visualize its efficiency.

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 easy search.

Evolution history [yyyyMMdd]

20130812 - Insert key overload for Master and Nested table, letting not to overwrite key if it already exists.
- Speeding up select operations and traversals with ValuesLazyLoadingIsOn.
20130811 -Remove KeyValue and get deleted value and notification if value existed in one round.
20130613 - Full locking of tables inside of transaction.
20130610 - Restoring table from the other table.
20130529 - Speeding up batch modifications (updates, random inserts) with Technical_SetTable_OverwriteIsNotAllowed instruction.
20121111 - Alternative tables storage locations.
20121101 - Added new iterators for transaction master and nested tables SelectForwardStartsWithClosestToPrefix and SelectBackwardStartsWithClosestToPrefix.
20121023 - DBreeze like in-memory database. “Out-of-the-box” bulk insert speed increase.
20121016 - Secondary Indexes. Going deeper. Part 2.
20121015 - Secondary Indexes. Going deeper.
20121012 - Behaviour of the iterators with the modification instructions inside.
20120922 - !!!! Important, attach new DBreeze and recompile your project, if you have errors concerning DateTime conversion functions - read the docu article.
- Storing virtual columns in the value, null-able datatypes and null-able text of fixed

length.
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.
(*updated 20121012*).
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 255 KB. Start using it by adding its reference to your project. Don't forget DBreeze.XML from 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 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 will 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();
```

After you have instantiated the engine two options will be 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 transaction after all necessary operations are done (using-statement makes it automatically).
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** to use try-catch block inside the transaction and **log exceptions** for the future analysis.

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

                }
                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 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** datatypes.

Note, that key in the table is always unique.

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

Keys 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

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?

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.

```
public void Example_InsertingData()
{
	using (var tran = engine.GetTransaction())
	{
		try
		{
			tran.Insert<int, int>("t1", 1, 1);
			tran.Commit();
		}
		catch (Exception ex)
		{
			Console.WriteLine(ex.ToString());
		}
	}
}
```

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, value type of 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 empty table will be created, if it doesn’t exist before.

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

                //NO COMMIT

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

We don’t store in the table data types, which you assume must be there, 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 then one table inside of the transaction.

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

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

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 fail can be the damage of the physical storage, and exceptions in the rollback procedure will bring 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 setup 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 table physical file bigger** but will take more time then one Commit after all operations.

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

                        //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();
                        }


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

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 execute 9 seconds.

If you start to insert data in random order it can take up to 3 minutes. That’s why, if you have in-memory big data set, before saving it to the database, sort it ascending in-memory by key and then insert. It will speed up your program.

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

Table operations. Partial Insert or Update

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

It’s not possible to save as a value byte array bigger then 2GB. For bigger data elements we will have to develop in the future other strategy.

In Dbreeze we have 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 as byte array.

DBreeze has special namespace inside, which allows you easily to 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 by us, if we think about the value as about the **set of columns** of the known length, like in **standard SQL databases**, and gives us ability to change every column separately, without changing all other 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 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())
        {
                try
                {
                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);

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

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 table has such value inside.

Row has **property Exists**:

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

So, if 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 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**.

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:

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

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

```
        }

}
```

A small benchmark for this procedure:
INSERT: 10361 ms; 28312951 ticks
FETCH: 4700 ms; 12844468 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 in 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 iterational 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 for filtering like this:**

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

Because it will work **much much much slower** then specially sharpened methods,
use 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 developer has always to take into consideration the idea of the 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 quantity of 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 non-existing table, this **table will no 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 **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 **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 de-serializer for objects. Slower then 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 de-serializer for objects. Much better then XML but not so good as serializer provided by http://json.codeplex.com/ - JSON.NET, though resides approximately the same physical space on the disk.

DbCustomSerializer<T> - gives you ability to attach your own serializer like http://json.codeplex.com/ .

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 de-serialization 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; }
}

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

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

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

```
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:

```
 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 writing thread releases its transaction, they will wait in a queue.

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

Note, if **one** of **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())
{
                try
                {
                        tran.SynchronizeTables("table1", "table2");
```

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

Transaction also has 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 user needs to know his bank account state, we don't need to block the table with account information, just read account state and return it. Doesn't matter that in this moment his account state is changing - it's a question of a moment. If user requests his account state in 5 minutes he will get 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 the 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 full object constructed then return it to the final projection as a ready element.

Note if you have received 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. If other thread removes value in between, after you have acquired the row, but still didn't acquired 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 behaviour for not synchronized read should be ok, because it's a question of the moment.

If you have acquired row and it exists, in one thread, now you are going to get the value, but in this moment other thread updates value, then you 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 other thread executes tran.RemoveAllKeys(“your reading table”, true - withFileRecreation). Your reading thread will get exception inside of the transaction. But all will be ok if other threads removes 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 constructor, after engine initialization, or for temporary tables, which are used for sub-computation with the help of database, and definitely only by one thread. For tables which are under read-write pressure, better to use tran.RemoveAll(false) and then one day to compact this table by copying existing values into new table, and renaming new table to old table.

Tables copying / compaction

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

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

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<Atricle>>(“Articles”)))
{
        ...
}
```

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

If you have mechanical HDD, its head must always move between two files to complete this operation, what 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 a key for 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;
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:

```
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

        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**

```
            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"

```
            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 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 defence 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 other 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 NoSql concept we have to have deals with many tables inside of one transaction, DBreeze has special constructions for tables 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 fo 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, excepts 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 how to store non-unique keys.

One of them is for every non-unique key create a separate table and store all reference 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 for the Data Access Layer, but gets back very fast responses from database.

Imagine that you have a plenty of Articles and every of it has 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 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.

```
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())
	{
		try
		{
		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);
		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: {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();
                }
                catch (Exception ex)
                {
                Console.WriteLine(ex.ToString());
                }
        }

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

        //Fetching data >=
        using (var tran = engine.GetTransaction())
        {
                try
                {
                //We are intereste 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());

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

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

        //Fetching data >
        using (var tran = engine.GetTransaction())
        {
                try
                {
                //We are intereste 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);
                }
                }
                }
                catch (Exception ex)
                {
                Console.WriteLine(ex.ToString());
                }
        }


        }
```

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

Articles<uint,DbMJSON<Article>>(“Articles”)

You remember that in namespace DBreeze.Utils there are a lot of extension 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 array.
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 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 tables structure.

We call it with a fancy word “fractal”, because it has 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 multi-dimensional storage concept.

It can also mean that in one value we can store object 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 make again 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 in values 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

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, lets assume we have master table with the name “t1”. Keys of this table are of integer type. Values can be different.

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

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

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 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 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 thread, who is using it, tries to modify table "t1", that's why all necessary techniques like tran.SynchronizeTables, if you modify more then one master table, must be used. They are described in previous chapters.

We have another method

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

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

Note, method **SelectTable** will not create 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 table name as 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 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 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 genearation
                .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 new method GetTable(uint tableIndex), where you can get 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 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)).

- Abillity 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.

We have a situation of memory growth in case if we use lot's 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 master table and all nested tables when transaction (working with master table) is finished. But only in case when parallel threads don't read from the same table in 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 the ways how to avoid this memory growth. It's not so critical when you insert in small chunks (100 records). So you must remember about 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 table, but it still stays in memory and make it growing. 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 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 NestedTable.Close, method, we want to close current table and all nested in it tables. Every master-table InsertTable or SelectTable (and nestedTable.GetTable) increase “open quantity” variable by 1, every CloseTable decreses value by 1, when value is less then 1, then the table with all nested in it tables will be closed. If we forget to close table then it will be open till all operations with master table are finished and automatic dispose 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 secondary index table we can store direct pointer 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 primary-table, who holds full information about the object, it's pointer can be moved, that's why our DAL must update value (pointer to the primary table key) in the secondary table also. When we delete from primary table, we must delete in the same transaction from 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 key and value. 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 behaviour for different threads like other structures.

Nested tables also have InsertDataBlock method.
Note, InsertDataBlock always return 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 column inside of a Row.

Note, if 2 parameter initialPointer is NULL then new data block will be created for the table, if not NULL it can mean that such 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 pointer to the data-block inside of “t1” key (17) starting from index 10, 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, optimized by all 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 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 table, 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 index. For example, in first table we store keys, like they are, with the content and in second table we store hashes of those keys and physical pointers to the first table. Now we can get 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 to use those 8 bytes or 16 bytes functions to stay collision-safe with a very high probability. If you need 1000% guarantee, use 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 property LinkToValue (actually it's a link to Key/Value), for getting 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 column 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 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 DBreeze constructor without configuration parameter at all.

If you have existing databases you can make its 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 folder for dbreeze incremental backup files “BackupFolderName”. That’s all.

If you start 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 to Write speed decrease, Read speed is untouched.

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

[20120922]

!!!!!!!!!!!!!!!!!!!!!!!! IMPORTANT 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, 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 thing.

Well, now what?
So, we have decided to leave DateTime converter to work with ulong. It doesn't have 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 this functions we recommend to work in the future**. The same algorithms are used by generic converter.

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 on the light way of the God's Love!

Storing in the value columns of the fixed size.

For the last some months we have created many tables with different value configurations, combining ways of the data storage. One of the most popular way is handling value byte[] as 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 then 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 of text in the 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 behaviour 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 byte array of 50+2 = **52 bytes** this you can store in your value from specific place (let's say 10).

Note, returned size will be always 2 bytes longer we need them to store 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 value. 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 setup a row version, then, depending upon this version, the further content of the value can have different configurations of the content.

[20121012]

Behaviour 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 example it will work good.

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 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 good. We don’t insert or delete “elements of the future iterations”.

In the next example we can have not desired behaviour:

```
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.

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. Enumerator will stop to iterate 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 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, so 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 modification list, we want to choose, synchronized or parallel. Default range selects, without extra parameter, 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 idea is 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 the 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 technique was described in previous articles. As a value we store physical pointer to the primary key inside of the “Article” table. When we have such physical pointer, searching of 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 new physical pointer into secondary index table).

If we remove Article, we must remove compound key from the table “ArticleIndexPrice” also. When we update price, inside of table Article, we must delete old compound key from the table “ArticleIndexPrice” and create 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 row with such Id exists before insert. If yes then we must read it, delete compound key, construct and insert new compound key into the table “ArticleIndexPrice” and finally update value in the table “Article”.

This all can slow down 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 inserts data, knows, if it’s going to be an update or new insert. That’s why Dbreeze can notify us about this.

We go on to insert data in 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 key/value pair, which was before modification and change 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 update. 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 **9300 ms (9 sec)**. **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)**. **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 in-memory database.

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

Instantiating example:

```
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 then .NET Dictionary or SortedDictionary, because has lots of sub-systems inside, which must be supported, and designed to work with very large data sets, without index fragmentation after continuous inserts, updates and deletes.

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

We have increased 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 buffer of size 1MB each, before disk flush. The 21-th (and so on, parallel) will write without buffer. After disposing of the writing transactions other tables can receive such 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, what 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 current DBreeze version we are able to set up tables 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 **value** of the Dictionary AlternativeTablesLocations key is **empty**, 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 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 corresponded 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 file will not be corrupted in case of power loss, first we have to write data into rollback file, then into 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 the random write. 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 degrade is not so visible. But we can see it very good when we need to update a batch of specific data.

We DON'T SEE SPEED DEGRADE, when we insert 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 case we should do nothing.

We CAN SEE SPEED DEGRADE, when we update 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 developer 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 table modification command.
- When new transaction starts, overwrite 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 batch ascending by key, before insert - it will economize HDD space.

Of course this technique makes 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 bring 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 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 there Microsoft has fixed bug with FileStream.Flush(true).

[20130608]

Restoring table from the other table.

Starting from DBreeze version 01.052 we can restore table 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.

Startign 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 https://dbreeze.codeplex.com/discussions/446373

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

[20130613]

Full tables locking inside of transaction.

Parallel threads can open transactions and in parallel read the same tables, in our standard configuration. For writing threads we use tran.SynchronizeTables command to sequentialize 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 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 transaction we want to define the lock type for the listed tables.

Note, we must use either 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()
        {
        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 transaction 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 previous chapter.

[20130811]

Remove KeyValue and get deleted value and notification if value existed 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 equals to true, will not give db to make an update.

Speeding up select operations and traversals with ValuesLazyLoadingIsOn.

Before DBreeze used lazy value loading technique. For example, we can say var row = transaction.Select<int,int>("t1",1);
at this moment we receive 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 known. It will be read out from DB only when we instruct row.Value.

Sometimes it is good when for us key is enough. Such cases can be when we store secondary index and link to the primary table is concatenated to the end of the key. Or if we have "multiple columns" in one value and we need to get only one column, and don't need to

get complete big value.

Nevertheless, lazy load will work a bit slower, in compare 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), but set it to false and all tran. traversal commands, like SelectForwards, Backwards etc., will return you row already with read out Value. So row.Value will read nothing from disk anymore, but return a value, sure if row.Exists = true.
This switch will also influent NestedTables which we get from tran.InsertTable, SelectTable and row.GetTable.
We can setup many time this switch inside of one transaction to tune different queries speed.

If something is not working like it is expected, please, don't hesitate to write down an issue comment on http://dbreeze.tiesky.com