DBreeze Database Benchmarking.

(dbreeze version 01.003.20120509; doc. version 01.002.20120509)

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

Testing Hardware

Processor:

Name: Intel Core I7 CPU 860 @ 2.8GHz
Max Clock Speed: 2794 MHz
Fron-Side Bus: 133 MHz
L2 Cache: 8192 kB
L3 Cach: 0 kB
Number of Cores: 4
Number of Logocal Processors: 8

OS:

Microsoft Windows 7 Ultimate SP1 64-bit

HDD:

WDC WD10EARS-00Y5B1 ATA Device
Total size: 931.5 GB
Total Cylinders: 121601
Total Sectors: 1953520065
Total Tracks: 31008255
Bytes per Sector: 512
Sectors per Track: 63
Tracks per Cylinder: 255

Benchmarking.

All tests source code you can find in the Solution, project VisualTester, class Benchmark.cs

Data Inserting. One Thread

Test 1 (private void TEST_1())

Bulk Insert
Inserting of 1MLN data with identity key where identity seed = 1. Commit after all inserts.

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

Quantity: 1 MLN
Insert took: 9.2 sec
Table file size: 18 MB

Test 2 (private void TEST_2())

Bulk Insert
Inserting of 1MLN / 100K / 10MLN data with the growing key. Commit after all inserts.

```
DateTime dt = new DateTime(1970, 1, 1);

for (int i = 0; i < 10000000; i++)
{
                tran.Insert<DateTime, byte[]>("t2", dt, null);
                dt = dt.AddSeconds(7);
}

tran.Commit();
```

Quantity: 1 MLN
Insert took: 9.9 sec
Table file size: 22 MB

Quantity: 100 000 (100K)
Insert took: 997 ms
Table file size: 2.2 MB

Quantity: 10 MLN

Insert took: 98 sec
Table file size: 220 MB

Test 3 (private void TEST_3())

*Inserting of 1MLN / 100K data with the growing key. Commit after **every** inserts.*

This type of test is not about the speed, but about the final table file size. Repeats standard accumulating data behaviour of the program.

```
DateTime dt = new DateTime(1970, 1, 1);

for (int i = 0; i < 1000000; i++)
{
                tran.Insert<DateTime, byte[]>("t2", dt, null);
                dt = dt.AddSeconds(7);
                tran.Commit();
}
```

Quantity: 1 MLN
Insert took: 212 sec
Table file size: 29 MB

Quantity: 100 000 (100K)
Insert took: 19 sec
Table file size: 2.9 MB

Test 3_2.1 (private void TEST_3_2())

*Updating of 1MLN / 100K data with the growing key. Commit after **every** update.*
(Initial data from TEST 3)

```
for (int i = 0; i < 1000000; i++)
{
                tran.Insert<DateTime, byte[]>("t2", dt, null);
                dt = dt.AddSeconds(7);
                tran.Commit();
}
```

Quantity: 1 MLN
Update took: 356 sec
Table file size: 29 MB

Quantity: 100 000 (100K)
Update took: 34 sec
Table file size: 2.9 MB

Test 3_2.2 (private void TEST_3_2())

Bulk update of 1MLN / 100K.
*Updating of 1MLN / 100K data with the growing key. Commit after **all** updates.*
(Initial data from TEST 3)

```
for (int i = 0; i < 1000000; i++)
{
                tran.Insert<DateTime, byte[]>("t2", dt, null);
                dt = dt.AddSeconds(7);
}
tran.Commit();
```

Quantity: 1 MLN
Update took: 28 sec
Table file size: 29 MB

Quantity: 100 000 (100K)
Update took: 2.8 sec
Table file size: 2.9 MB

Test 7 (private void TEST_7())

Bulk insert of Random Keys of 1MLN / 100K.
*Insert of 1MLN / 100K data with the **random** key. Commit after **all inserts**.*

***Note**,* this test is not only for speed calculation, but for checking Table File size.
Here we emulating filling the table with random 4 bytes keys.
For bulk inserts better to use preliminary sorted (in memory) data in ascending order, to make application more performant.

```
int vl = 0;
Random rnd = new Random();
for (int i = 0; i < 1000000; i++)
{
        vl = rnd.Next(1000000);

        tran.Insert<int, byte[]>("t5", vl, null);

}


tran.Commit();
```

Quantity: 1MLN
Insert took: 330 sec
Table file size: 21 MB

Quantity: 100 000 (100K)
Insert took: 25 sec
Table file size: 2.9 MB

Test 7.1
*Insert of 200K data with the **random** key. Commit after **every insert**.*

Quantity: 200 000 (200K)
Insert took: 67 sec
Table file size: 5.5 MB

Data Fetching

Fetch tests will be based on such insert:

```
DateTime dt = new DateTime(1970, 1, 1);

                for (int i = 0; i < 10000000; i++)
{
                tran.Insert<DateTime, byte[]>("t2", dt, null);
                dt = dt.AddSeconds(7);
}

tran.Commit();
```

We got 10 MLN of 8 bytes keys with a “good jump”

Test 8.1 (based private void TEST_8())
*Iterating **forward** from 1 key, taking 1MLN / 100K / 10K / 1K*

```
int cnt = 0;
foreach (var row in tran.SelectForward<DateTime, byte[]>("t2").Take(1000000))
{
                cnt++;
}
```

Select 1MLN took: 6 sec (6055 ms)
Select 100K took: 638 ms
Select 10K took: 81 ms
Select 1K took: 22 ms

Test 8.2 (based private void TEST_8())
*Iterating **forward** from 1 key, taking 1MLN / 100K / 10K / 1K, also **taking value***

```
int cnt = 0;
byte[] val=null;
foreach (var row in tran.SelectForward<DateTime, byte[]>("t2").Take(1000000))
{
                    val = row.Value;
                    cnt++;
}
```

Select 1 MLN took: 10 sec (10581 ms)
Select 100K took: 1165 ms
Select 10K took: 130 ms
Select 1K took: 29 ms

Test 8.3 (based private void TEST_8())
*Iterating **backward** from 1 key, taking 1MLN / 100K / 10K / 1K*

```
int cnt = 0;
foreach (var row in tran.SelectForward<DateTime, byte[]>("t2").Take(1000000))
{
                    cnt++;
}
```

Select 1MLN took: 6200 ms
Select 100K took: 630 ms
Select 10K took: 81 ms
Select 1K took: 23 ms

Test 8.4 (based private void TEST_8())
*Iterating **backward** from 1 key, taking 1MLN / 100K / 10K / 1K, also **taking value***

```
int cnt = 0;
byte[] val=null;
foreach (var row in tran.SelectForward<DateTime, byte[]>("t2").Take(1000000))
{
                    val = row.Value;
                    cnt++;
}
```

Select 1 MLN took: 10501 ms
Select 100K took: 1070 ms
Select 10K took: 130 ms
Select 1K took: 29 ms

Test 8.5 (based private void TEST_8_5())
*Iterating **forward from** the **key N**, taking 100K*
*Iterating **backward from** the **key N**, taking 100K - takes the same time as forward*

10 MLN keys contain DateTime from 01.01.1970 - 21.03.1972.
We will make 3 test, key is starting from
~25% - 01.07.1970
~50% - 01.06.1971
~75% - 01.01.1972

```
int cnt = 0;
byte[] val = null;
//dtSearch we will change for every test
DateTime dtSearch = new DateTime(1970,7,1);
//dtSearch = new DateTime(1971, 6, 1);
//dtSearch = new DateTime(1972, 1, 1);

foreach (var row in tran.SelectForwardStartFrom<DateTime, byte[]>("t2",
dtSearch,true).Take(100000))
{
        val = row.Value;
        cnt++;
}
```

*Select 100K **~25%** took: 1152 ms*
*Select 100K **~50%** took: 1102 ms*
*Select 100K **~75%** took: 1098 ms*

Test 8.6 (based private void TEST_8_6())
*Iterating **forward from** the **key N to key M, with value acquiring***

In here every found key must be compared with To Key before returning

10 MLN keys contain DateTime from 01.01.1970 - 21.03.1972.
We will make 3 test, key is starting from
~25% - 01.07.1970
~50% - 01.06.1971
~75% - 01.01.1972

```
int cnt = 0;
byte[] val = null;

DateTime dtSearch = new DateTime(1970, 7, 1);
//dtSearch = new DateTime(1971, 6, 1);
//dtSearch = new DateTime(1972, 1, 1);
DateTime dtSearchStop = dtSearch.AddMonths(1);

foreach (var row in tran.SelectForwardFromTo<DateTime, byte[]>("t2", dtSearch, true,
dtSearchStop,true))
{
                val = row.Value;
                cnt++;
}
```

*Select 100K **~25%** took: 8542 ms   Returned count: 382628*

*Select 100K **~50%** took: 6092 ms Returned count: 370286*
*Select 100K **~75%** took: 4430 ms Returned count: 382629*

*Select 100K **~25%** took: 1116 ms .Take(100000)*
*Select 100K **~50%** took: 1116 ms .Take(100000)*
*Select 100K **~75%** took: 1096 ms .Take(100000)*

Test 8.7 (based private void TEST_8_7())
*Iterating **backward from** the **key N to key M, with value acquiring***

In here every found key must be compared with To Key before returning

10 MLN keys contain DateTime from 01.01.1970 - 21.03.1972.
We will make 3 test, key is starting from
~25% - 01.07.1970
~50% - 01.06.1971
~75% - 01.01.1972

```
int cnt = 0;
byte[] val = null;

DateTime dtSearch = new DateTime(1970, 7, 1);
//dtSearch = new DateTime(1971, 6, 1);
//dtSearch = new DateTime(1972, 1, 1);
DateTime dtSearchStop = dtSearch.AddMonths(-1);

foreach (var row in tran.SelectBackwardFromTo<DateTime, byte[]>("t2", dtSearch, true,
dtSearchStop,true))
{
                val = row.Value;
                cnt++;
}
```

*Select 100K **~25%** took: 5060 ms Returned count: 370286*
*Select 100K **~50%** took: 7500 ms Returned count: 382629*
*Select 100K **~75%** took: 9081 ms Returned count: 382628*

*Select 100K **~25%** took: 1111 ms .Take(100000)*
*Select 100K **~50%** took: 1131 ms .Take(100000)*
*Select 100K **~75%** took: 1125 ms .Take(100000)*

Test 8.8 (based private void TEST_8_8())

*SkipForward from start 3MLN, then take 100000, **with values acquiring (skipped keys don't acquire values)***

```
foreach (var row in tran.SelectForwardSkip<DateTime, byte[]>("t2", 3000000).Take(100000))
```

```
{
                val = row.Value;
                cnt++;
}
```

SkipSelect 3MLN / 100K took: 3202 ms

SkipForward from start 6MLN, then take 100000, ***with values acquiring***
SkipSelect 6MLN / 100K took: 5605 ms

SkipForward from start 6MLN, then take 100000, ***with values acquiring***
SkipSelect 9MLN / 100K took: 7828 ms

*Skip****Backward*** *from start 3MLN, then take 100000,* ***with values acquiring***
SkipSelect 3MLN / 100K took: 3283 ms

*Skip****Backward*** *from start 6MLN, then take 100000,* ***with values acquiring***
SkipSelect 6MLN / 100K took: 5545 ms

*Skip****Backward*** *from start 3MLN, then take 100000,* ***with values acquiring***
SkipSelect 9MLN / 100K took: 7876 ms

*Skip****Backward*** *from start 3MLN, then take 100000,* ***without value acquiring***
SkipSelect 9MLN / 100K took: 7451 ms

Test 8.9 (based private void TEST_8_9())
SkipForwardFromKey, skip 100K / 1MLN, then take 100000, ***with values acquiring (skipped keys don't acquire values)***

```
 int cnt = 0;
                byte[] val = null;
                DateTime dtSearch = new DateTime(1970, 7, 1);
                //dtSearch = new DateTime(1971, 6, 1);
                //dtSearch = new DateTime(1972, 1, 1);
                foreach (var row in
                tran.SelectForwardSkipFrom<DateTime, byte[]>("t2", dtSearch,
100000).Take(100000))
                {
                        val = row.Value;
                        cnt++;
                }
```

SkipForwardFromKey ~25% key skip 100K the take 100 K ***with values acquiring***
took: 1199 ms

*SkipForwardFromKey ~25% key skip 1MLN the take 100 K **with values acquiring***
took: 1825 ms

*SkipForwardFromKey ~50% key skip 100K the take 100 K **with values acquiring***
took: 1155 ms

*SkipForwardFromKey ~50% key skip 1MLN the take 100 K **with values acquiring***
took: 1809 ms

*SkipForwardFromKey ~75% key skip 100K the take 100 K **with values acquiring***
took: 1149 ms

*SkipForwardFromKey ~75% key skip 1MLN the take 100 K **with values acquiring***
took: 733 ms

*SkipBackwardFromKey, skip 100K / 1MLN, then take 100000, **with values acquiring (skipped keys don't acquire values)***

*SkipBackwardFromKey ~25% key skip 100K the take 100 K **with values acquiring***
took: 1171 ms

*SkipBackwardFromKey ~25% key skip 1MLN the take 100 K **with values acquiring***
took: 1805 ms

*SkipBackwardFromKey ~50% key skip 100K the take 100 K **with values acquiring***
took: 1155 ms

*SkipBackwardFromKey ~50% key skip 1MLN the take 100 K **with values acquiring***
took: 1786 ms

*SkipBackwardFromKey ~75% key skip 100K the take 100 K **with values acquiring***
took: 1206 ms

*SkipBackwardFromKey ~75% key skip 1MLN the take 100 K **with values acquiring***
took: 1786 ms

Test 9 (based private void TEST_1_9())
*Selecting random keys 10K/ 100K / 1MLN, **with and without values acquiring***

This test is built based on Test 1 (where we have ints from 1 to 1000000) all random search keys exist in this example

```
Random rnd = new Random();
int key = 0;
byte[] val = null;
```

```
for (int i = 0; i < 100000; i++)
{
	key = rnd.Next(999999);
	var row = tran.Select<int, byte[]>("t1", key);

	if (row.Exists)
			val = row.Value;  //or remarked if without values (just keys)

}
```

Random Select 10K took: 1234 ms with value acquiring
Random Select 10K took: 1199 ms without value acquiring

Random Select 100K took: 12177 ms with value acquiring
Random Select 100K took: 11726 ms without value acquiring

Random Select 1MLN took: 122577 ms with value acquiring
Random Select 1MLN took: 116120 ms without value acquiring

Data Inserting. Multi-Threading

In this test we will write data from many threads in different way.

Test 10.1 (based private void TEST_3_1_STARTER_2())

```
DateTime dt = new DateTime(1970, 1, 1);
for (int i = 0; i < 10000; i++)
{
	tran.Insert<DateTime, byte[]>(tableName, dt, null);
	dt = dt.AddSeconds(7);
	//tran.Commit();
}

tran.Commit();
```

* We have empty db and then start 100 threads, which have to write into their own table 1K (1000) records and **Commit of every table after every insert**

The whole operation was finished in 7229 ms, we got 100 tables and 1000 records in every of it *(13.8K records/second)*

* We have empty db and then start 100 threads, which have to write into their own table 10K

(10000) records and **Commit of every table after every insert**

The whole operation was finished in 87229 ms (87 sec), we got 100 tables and 10000 records in every of it (12K records/second)

* We have empty db and then start 100 threads, which have to write into their own table 1K (1000) records and **Commit of every table after all records are inserted in this table**

The whole operation was finished in 506 ms, we got 100 tables and 1000 records in every of it (200K records/second)

* We have empty db and then start 100 threads, which have to write into their own table 10K (10000) records and **Commit of every table after all records are inserted in this table**

The whole operation was finished in 4043 ms, we got 100 tables and 10000 records in every of it (247K records/second)

* We have empty db and then start 1 threads, which have to write into their own table 1MLN (1000000) records and **Commit of every table after all records are inserted in this table**

The whole operation was finished in 9887 ms, we got 1 tables and 100000 records in it (102K records/second)

* We have empty db and then start 3 threads, which have to write into their own table 1MLN (1000000) records and **Commit of every table after all records are inserted in this table**

The whole operation was finished in 10702 ms, we got 3 tables and 100000 records in every of it
(300K records/second)

* We have empty db and then start 6 threads, which have to write into their own table 1MLN (1000000) records and **Commit of every table after all records are inserted in this table**

The whole operation was finished in 18290 ms, we got 3 tables and 100000 records in every of it
(333K records/second)

After this test we have 6 files 22MB each, in total 132 MB. Writing speed was 7.3 MB/s and it was random writing speed.

Multiple tables.

This test is dedicated to the efficiency of working with multiple tables. For now every table

locates in 3 OS files. One for pure data, second for rollback data and third for efficient rollback usage. And this test was done under NTFS (which uses b+tree for locating files and files fragments).
We will try in following tests to create, to write and to read data from 34K/ 200K/1 MLN tables.

Multiple tables. Write/Read.

Test 11 (based private void TEST_9() - write and TEST_9_3() - read)

Database is empty to the beginning of the test. We will use such script to create and to write a small piece of data into 34K/ 200K/1 MLN tables:

```
//This value will change from test to test
int totalTablesToCreate = 1000000;

 for (int i = 1; i <= totalTablesToCreate; i++)
 {
                using (var tran = engine.GetTransaction())
                {
                        try
                        {

                                tran.Insert<byte[], byte[]>("t" + i, new byte[] { 0 }, new byte[] { 0 });
                                tran.Commit();

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

}
```

and then we used such script to read data from all these tables:

```
//This value will change from test to test
int totalTablesToCreate = 1000000;
byte[] v = null;

for (int i = 1; i <= totalTablesToCreate; i++)
{

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

```
                    var row = tran.Select<byte[], byte[]>("t" + i, new byte[] { 0 });
                    v = row.Value;
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }

        }
}
```

Results:

Write + access time:

34K tables - 33 sec ~ 1030 tables per second
200K tables - 263 sec ~ 760 tables per second
1MLN tables - 1360 sec ~ 735 tables per second

Filesystem information for 1MLN tables:
Quantity of files in DBreeze instance folder - 3.000.006 (together with scheme and transaction journal)
Total folder size: 140 MB
Total size which folder resides on the physical disk (depends upon cluster size): 7,67 GB

Read + access time:

34K tables - 15 sec ~ 2266 tables per second
200K tables - 121 sec ~ 1652 tables per second
1MLN tables - 736 sec ~ 1358 tables per second

Memory:

Write test, project started from 28 MB and finished with 33 MB in RAM.
Read test, project started from 28 MB and finished with 28 MB in RAM.