RaptorDB - The Key Value Store V2

Mehdi Gholam, 11 Oct 2013

Even faster Key/Value store nosql embedded database engine utilizing the new MGIndex data structure with MurMur2 Hashing and WAH Bitmap indexes for duplicates. (with MonoDroid support)

Download RaptorDB_v2.0.zip - 38.7 KB
Download RaptorDB_v2.1.zip - 39 KB
Download RaptorDB_v2.2.zip - 39 KB
Download RaptorDB_v2.3.zip - 39.6 KB
Download RaptorDB_v2.4.zip - 39.9 KB
Download RaptorDB_v2.5.zip
Download RaptorDB_v2.6.zip
Download RaptorDB_v2.7.0.zip
Download RaptorDB_v2.7.5.zip

Introduction

This article is the version 2 of my previous article found here (http://www.codeproject.com/Articles/190504/RaptorDB), I had to write a new article because in this version I completely redesigned and re-architected the original and so it would not go with the previous article. In this version I have done away with the b+tree and hash index in favor of my own `MGIndex` structure which for all intents and purposes is superior and the performance numbers speak for themselves.

What is RaptorDB?

Here is a brief overview of all the terms used to describe `RaptorDB`:

- **Embedded**: You can use `RaptorDB` inside your application as you would any other DLL, and you don't need to install services or run external programs.
- **NoSQL**: A grass roots movement to replace relational databases with more relevant and specialized storage systems to the application in question. These systems are usually designed for performance.
- **Persisted**: Any changes made are stored on hard disk, so you never lose data on power outages or crashes.
- **Dictionary**: A key/value storage system much like the implementation in .NET.
- **MurMurHash**: A non cryptographic hash function created by Austin Appleby in 2008 (http://en.wikipedia.org/wiki/MurmurHash).

Features

`RaptorDB` has the following features :

- Very fast performance (typically 2x the insert and 4x the read performance of `RaptorDB` v1)
- Extremely small foot print at ~50kb.
- No dependencies.
- Multi-Threaded support for read and writes.
- Data pages are separate from the main tree structure, so can be freed from memory if needed, and loaded on demand.
- Automatic index file recovery on non-clean shutdowns.
- String Keys are UTF8 encoded and limited to 60 bytes if not specified otherwise (maximum is 255 chars).
- Support for long string Keys with the `RaptorDBString` class.
- Duplicate keys are stored as a WAH Bitmap Index for optimal storage and speed in access.
- Two mode of operation Flush immediate and Deferred (the latter being faster at the expense of the risk of non-clean shutdown data loss).
- Enumerate the index is supported.
- Enumerate the Storage file is supported.
- Remove Key is supported.

Why another data structure?

There is always room for improvement, and the ever need for faster systems compels us to create new methods of doing things.

```
MGindex
```

is no exception to this rule. Currently `MGindex` outperforms b+tree by a factor of **15x on writes** and **21x on reads**, while keeping the main feature of disk friendliness of a b+tree structure.

The problem with a b+tree

Theoretically a b+tree is O(N log k N) or log base k of N, now for the typical values of k which are above 200 for example the b+tree should outperform any binary tree because it will use less operations. However I have found the following problems which hinder performance :

- Pages in a b+tree are usually implemented as a list or array of child pointers and so while finding and inserting a value is a O(log k) operation the process actually has to move children around in the array or list, and so is time consuming.
- Splitting a page in b+tree has to fix parent nodes and children so effectively will lock the tree for the duration, so parallel updates are very very difficult and have spawned a lot of research articles.

Requirements of a good index structure

So what makes a good index structure, here are what I consider essential features of one:

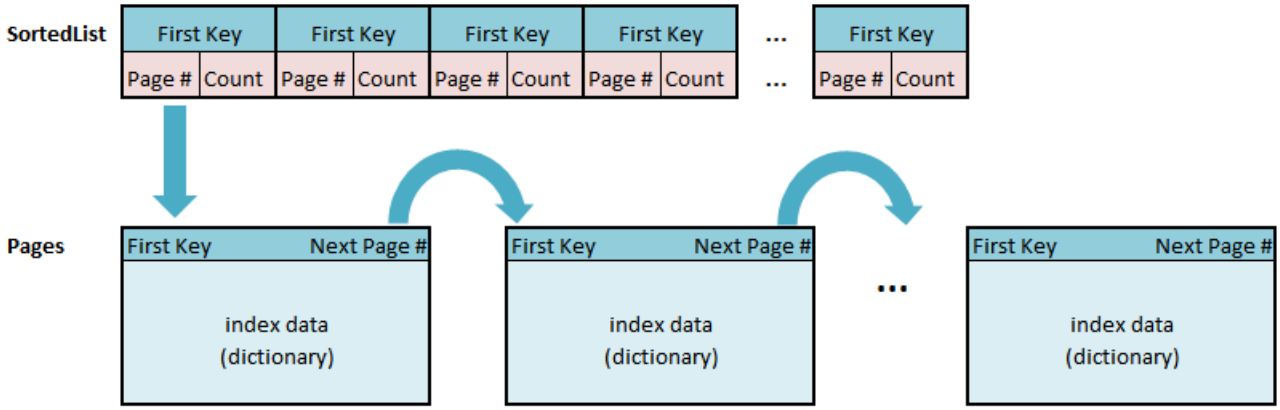
- Page-able data structure:
  - Easy loading and saving to disk.
  - Free memory on memory constraints.

    - On-demand loading for optimal memory usage.

- Very fast insert and retrieve.
- Multi-thread-able and parallel-able usage.
- Pages should be linked together so you can do range queries by going to the next page easily.

The MGIndex

`MGIndex` takes the best features of a b+tree and improves upon on them at the same time removing the impediments. `MGIndex` is also extremely simple in design as the following diagram shows:



As you can see the page list is a sorted dictionary of first keys from each page along with associated page number and page items count. A page is a dictionary of key and record number pairs.
This format ensures a semi sorted key list, in that within a page the data is not sorted but pages are in sort order relative to each other. So a look-up for a key just compares the first keys in the page list to find the page required and gets the key from the page's dictionary.

`MGIndex` is O(log M)+O(1), M being N / PageItemCount [`PageItemCount` = 10000 in the `Globals` class]. This means that you do a binary search in the page list in log M time and get the value in O(1) time within a page.

`RaptorDB` starts off by loading the page list and it is good to go from there and pages are loaded on demand, based on usage.

Page Splits

In the event of page getting full and reaching the `PageItemCount`,

```
MGIndex
```

will sort the keys in the page's dictionary and split the data in two pages (similar to a b+tree split) and update the page list by adding the new page and changing the first keys needed. This will ensure the sorted page progression.

Interestingly the processor architecture plays an important role here as you can see in the performance tests as it is directly related to the sorting key time, the Core iX processors seem to be very good in this regard.

Interesting side effects of MGIndex

Here are some interesting side effects of `MGIndex`

- Because the data pages are separate from the Page List structure, implementing locking is easy and isolated within a page and not the whole index, not so for normal trees.
- Splitting a page when full is simple and does not require a tree traversal for node overflow checking as in a b+tree.
- Main page list updates are infrequent and hence the locking of the main page list structure does not impact performance.
- The above make the `MGIndex` a really good candidate for parallel updates.

The road not taken / the road taken and doubled back!

Originally I used a `AATree` found here (http://demakov.com/snippets/aatree.html) for the page structures, for being extremely good and simple structure to understand. After testing and comparing to the internal .net `SortedDictionary` (which is a Red-Black tree structure) it was slower and so scrapped (see the performance comparisons).

I decided against using `SortedDictionary` for the pages as it was slower than a normal `Dictionary` and for the purpose of a key value store the sorted-ness was not need and could be handled in other ways. You can switch to the `SortedDictionary` in the code at any time if you wish and it

makes no difference to the overall code other than you can remove the sorting in the page splits.

I also tried an assorted number of sorting routines like double pivot quick sort, timsort, insertion sort and found that they all were slower than the internal .net quicksort routine in my tests.

Performance Tests

In this version I have compiled a list of computers which I have tested on and below is the results.

Item count : 10,000,000 Times in seconds

Computer	Specifications	Set time	Get time	Sets/sec	Gets/sec	Comments
Lenovo U165	AMD K625 1.6ghz,4gb Ram, 5400rpm, win7 64	236	119	42,373	84,034	My own notebook
Acer S3	Core i3 1.3ghz, 4gb, 5400rpm, win7 64	134	151	74,627	66,225	Lower clock to my system but superior architecture
HP Pavillion G4	Core i5 2.4ghz, 4gb, 5400rpm, win7 64	53	98	188,679	102,041	
HP Pavillion DV3	Core i5 2.4ghz, 4gb, 5400rpm, win7 64	78	57	128,205	175,439	
Test Server 1	Core2 Quad 2.4ghz, 8gb, 5400 raid5, win2008 r2 64	85	89	117,647	112,360	Hand built
Test Server 2	Core2 Quad q6600 2.4ghz, 8gb, 5400 raid5, win2008 r2 64	90	98	111,111	102,041	Hand built
HP ML120G6	Xeon X3430, 12gb, 10K raid5, win2008 r2	53	63	188,679	158,730	

As you can see you get a very noticeable performance boost with the new Intel Core iX processors.

Comparing B+tree and MGIndex

For a measure of relative performance of a b+tree, Red/Black tree and `MGIndex` I have compiled the following results.

Pure Index speed 1,000,000

Structure Type	Set (s)	Get (s)
B+tree	9.1	8.2
SortedDictionary	4.4	2.7
MGIndex	0.6	0.3

Times are in seconds.

B+Tree : is the index code from `RaptorDB` v1
SortedDictionary : is the internal .net implementation which is said to be a Red/Black tree.

Really big data sets!

To really put the engine under pressure I did the following tests on huge data sets (times are in seconds, memory is in Gb) :

Count	Set (s)	Get(s)	Set/sec	Get/sec	Memory	Comments
10,000,000	58	63	172,414	158,730	2	
20,000,000	167	132	119,760	151,515	3.7	
50,000,000	744	370	67,204	135,135	6.7	
100,000,000	1978	NT	50,556	NT	11.7	excluding array of guids
20,000,000	314	212	63,694	94,340	-	RaptorDB v1.8 B+Tree

These tests were done on a HP ML120G6 system with 12Gb Ram, 10k raid disk drives running Windows 2008 Server R2 64 bit. For a measure of relative performance to `RaptorDb` v1 I have included a 20 million test with that engine also.

I deferred from testing the get test over 100 million record as it would require a huge array in memory to store the `Guid` keys for finding later, that is why there is a NT (not tested) in the table.

Interestingly the read performance is relatively linear.

Index parameter tuning

To get the most out of `RaptorDB` you can tune some parameters specific to your hardware.

- `PageItemCount` : controls the size of each page.

Here are some of my results:

MGIndex : 1,000,000 items

Node Count	Split time (s)	Pages	Set (s)	Get (s)
30000	1.95	64	10.13	10.1
20000	1.48	64	9.43	13.33
10000	1.23	128	9.28	10.65
5000	1.16	256	9.54	10.97
1000	1.3	1380	8.31	10.6

I have chosen the 10000 number as a good case in both read and writes, you are welcome to tinker with this on your own systems and see what works better for you.

Performance Tests v2.3

In v2.3 a single simple change of converting internal classes to structs rendered huge performance improvements of 2x+ and at least 30% lower memory usage. You are pretty much guaranteed to get **100k+ insert performance** on any system.

Item count : 10,000,000 Times in seconds

Computer	Specifications	Set time	Get time	Sets/sec	Gets/sec	Comments
Lenovo U165	AMD K625 1.6ghz,4gb Ram, 5400rpm, win7 64	61	123	163,934	81,301	My own notebook
Acer S3	Core i3 1.3ghz, 4gb, 5400rpm, win7 64	90	95	111,111	105,263	test #1
Acer S3	same as above	45	78	222,222	128,205	test #2
Acer S3	same as above	45	71	222,222	140,845	test #3
HP Pavillion G4	Core i5 2.4ghz, 4gb, 5400rpm, win7 64	50	49	200,000	204,082	test #1
HP Pavillion G4	same as above	24	75	416,667	133,333	test #2 : incredible insert speed
HP Pavillion G4	same as above	33	47	303,030	212,766	test #3
Test Server 1	Core2 Quad 2.4ghz, 8gb, 5400 raid5, win2008 r2 64	36	84	277,778	119,048	Hand built
Test Server 2	Core2 Quad q6600 2.4ghz, 8gb, 5400 raid5, win2008 r2 64	41	95	243,902	105,263	Hand built
Test Server 3	AMD FX4100 Quad 3.6Ghz, 8gb, win2008 r2 64	32	80	312,500	125,000	Hand built

Some of the test above were run 3 times because the computers were being used at the time (not cold booted for the tests) so the initial results were off. The HP G4 laptop is just astonishing.

I also re-ran the 100 million test on the last server in the above list and here is the results:

Count	Set (s)	Get(s)	Set/sec	Get/sec	Memory	Comments
100,000,000	508	NT	196,850	NT	5.6	excluding array of guids

As you can see in the above test, the insert time is 4x faster (although the computer specs to not match the HP system tested earlier) and incredibly the memory usage is half than the previous test.

Using the Code

To create or open a database you use the following code :

```
// to create a db for guid keys without allowing duplicates
var guiddb = RaptorDB.RaptorDB<Guid>.Open("c:\\RaptorDbTest\\multithread", false);

// to create a db for string keys with a length of 100 characters (UTF8) allowing duplicates
var strdb = RaptorDB.RaptorDB<string>.Open("c:\\intdb", 100, true);
```

To insert and retrieve data you use the following code :

```
Guid g = Guid.NewGuid();
guiddb.Set(g, "somevalue");

string outstr="";
if(guiddb.Get(g, out outstr))
{
   // success outstr should be "somevalue"
}
```

The UnitTests project contains working example codes for different use cases so you can refer to it for more samples.

Differences to v1

The following are a list of differences in v2 opposed to v1 of `RaptorDB`:

- Log Files have been removed and are not needed anymore as the `MGIndex` is fast enough for in-process indexing.
- Threads have been replaced by timers.
- The index will be saved to disk in the background without blocking the engine process.
- Messy generic code has been simplified and the need for a `RDBDataType` has been removed, you can use normal int, long, string and Guid data types.
- `RemoveKey` has been added.

Other than that existing code should compile as is with the new engine.

Using RaptorDBString and RaptorDBGuid

`RaptorDBString` is for long string keys (larger than 255 characters) and it is really useful for file paths etc. You can use it in the following way :

```
// long string keys without case sensitivity
var rap = new RaptorDBString(@"c:\raptordbtest\longstringkey", false);

// murmur hashed guid keys
var db = new RaptorDBGuid("c:\\RaptorDbTest\\hashedguid");
```

`RaptorDBGuid` is a special engine which will `MurMur2` hash the input `Guid` for lower memory usage (4 bytes opposed to 16 bytes), this is useful if you have a huge number of items which you need to store. You can use it in the following way :

```
// murmur hashed guid keys
var db = new RaptorDBGuid("c:\\RaptorDbTest\\hashedguid");
```

Global parameters

The following parameters are in the Global.cs file which you can change which control the inner workings of the engine.

Parameter	Default	Description
<code>BitmapOffsetSwitchOverCount	10	Switch over point where duplicates are stored as a WAH bitmap opposed to a list of record numbers
<code>PageItemCount	10,000	The number of items within a page
<code>SaveTimerSeconds	60	Background save index timer seconds (e.g. save the index to disk every 60 seconds)
<code>DefaultStringKeySize	60	Default string key size in bytes (stored as UTF8)
<code>FlushStorageFileImmetiatley	false	Flush to storage file immediately
<code>FreeBitmapMemoryOnSave	false	Compress and free bitmap index memory on saves

RaptorDB interface

`Set(T, byte[])`	Set Key and byte array Value, returns void
`Set(T, string)`	Set Key and string Value, returns void

`Get(T, out string)`	Get the Key and put it in the string output parameter, returns true if key was found
`Get(T, out byte[])`	Get the Key and put it in the byte array output parameter, returns true if key was found
`<code>RemoveKey(T)`	This will remove the key from the index
`EnumerateStorageFile()`	returns all the contents of the main storage file as an `IEnumerable< KeyValuePair<T, byte[]> >`
`Enumerate(fromkey)`	Enumerate the Index from the key given.
`GetDuplicates(T)`	returns a list of main storage file record numbers as an `IEnumerable<int>` of the duplicate key specified
`FetchRecord(int)`	returns the Value from the main storage file as `byte[]`, used with `GetDuplicates` and `Enumerate`
`Count(includeDuplicates)`	returns the number of items in the database index , counting the duplicates also if specified
`SaveIndex()`	Allows the immediate save to disk of the index (the engine will automatically save in the background on a timer)
`Shutdown()`	This will close all files and stop the engine.

Non-clean shutdowns

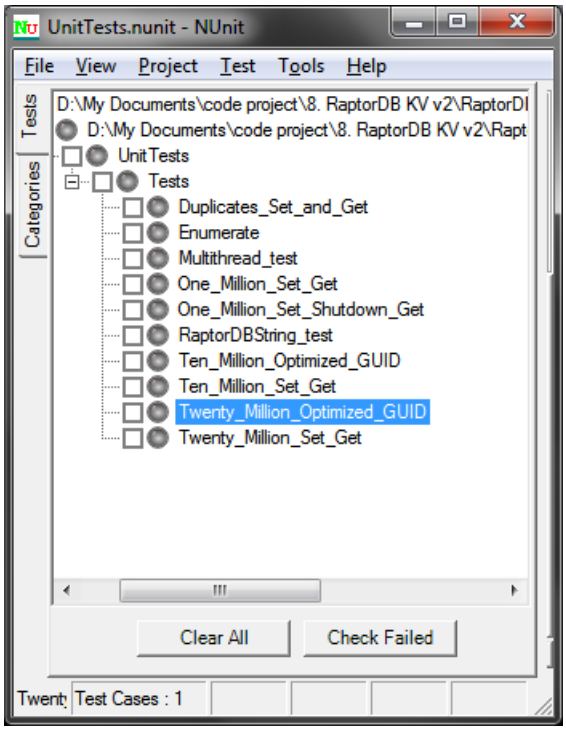
In the event of a non clean shutdown `RaptorDB` will automatically rebuild the index from the last indexed item to the last inserted item in the storage file. This feature also enables you to delete the mgidx file and have `RaptorDB` rebuild the index from scratch.

Removing Keys

In v2 of `RaptorDB` removing keys has been added with the following caveats :

- Data is **not** deleted from the storage file.
- A special delete record is added to the storage file for tracking deletes and which also help with index rebuilding when needed.
- Data **is** removed from the index.

Unit Tests



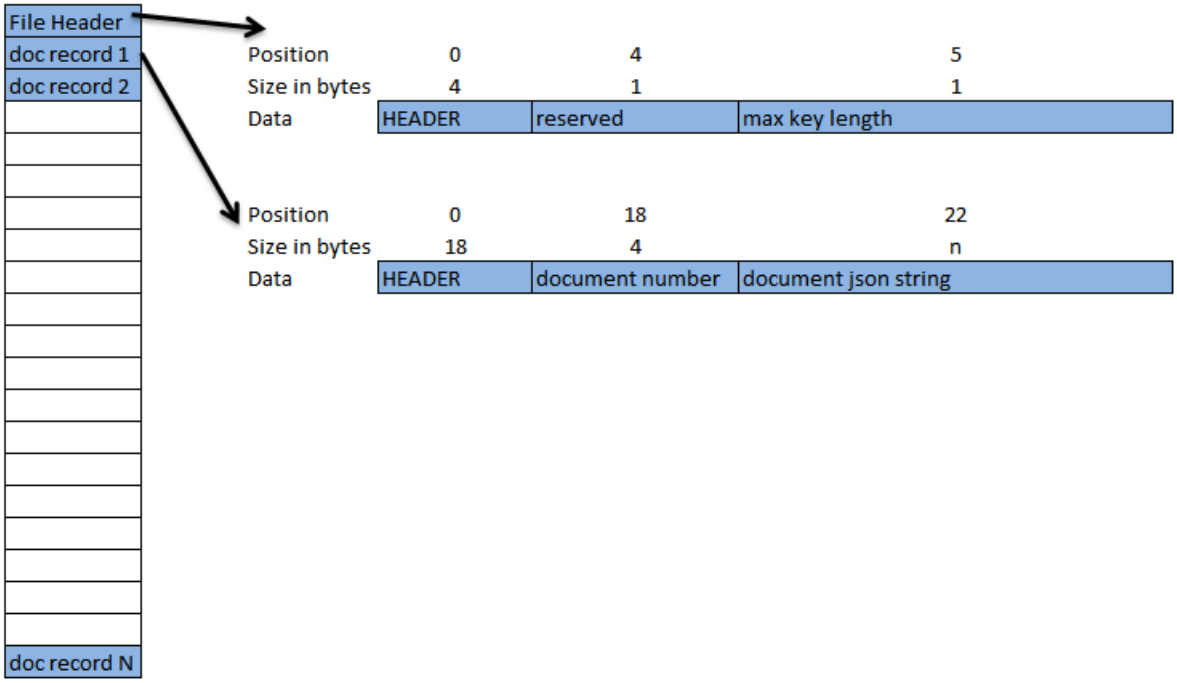
The following unit tests are included in the source code (the output folder for all the tests is `C:\RaptorDbTest`):

- **Duplicates_Set_and_Get** : This test will generate 100 duplicates of 1000 `Guid`s and fetch each one (This tests the WAH bitmap subsystem).
- **Enumerate** : This test will generate 100,001 `Guid`s and enumerate the index from a predetermined `Guid` and show the result count (the count will differ between runs).
- **Multithread_test** : This test will create 2 threads inserting 1,000,000 items and a third thread reading 2,000,000 items with a delay of 5 seconds from the start of insert.
- **One_Million_Set_Get** : This test will insert 1,000,000 items and read 1,000,000 items.
- **One_Million_Set_Shutdown_Get** : This test will do the above but shutdown and restart before reading.
- **RaptorDBString_test** : This test will create 100,000 1kb string keys and read them from the index.
- **Ten_Million_Optimized_GUID** : This test will use the `RaptorDBGuid` class which will MurMur hash 10,000,000 `Guid`s writting and reading them.
- **Ten_Million_Set_Get** : The same as 1 million test but with 10 million items.
- **Twenty_Million_Optimized_GUID** : The same as 10 million test but with 20 million items.
- **Twenty_Million_Set_Get** : The same as 1 million test but with 20 million items.
- **StringKeyTest** : A test for normal string keys of max 255 length.
- **RemoveKeyTest** : A test for removing keys works properly between shutdowns.

File Formats

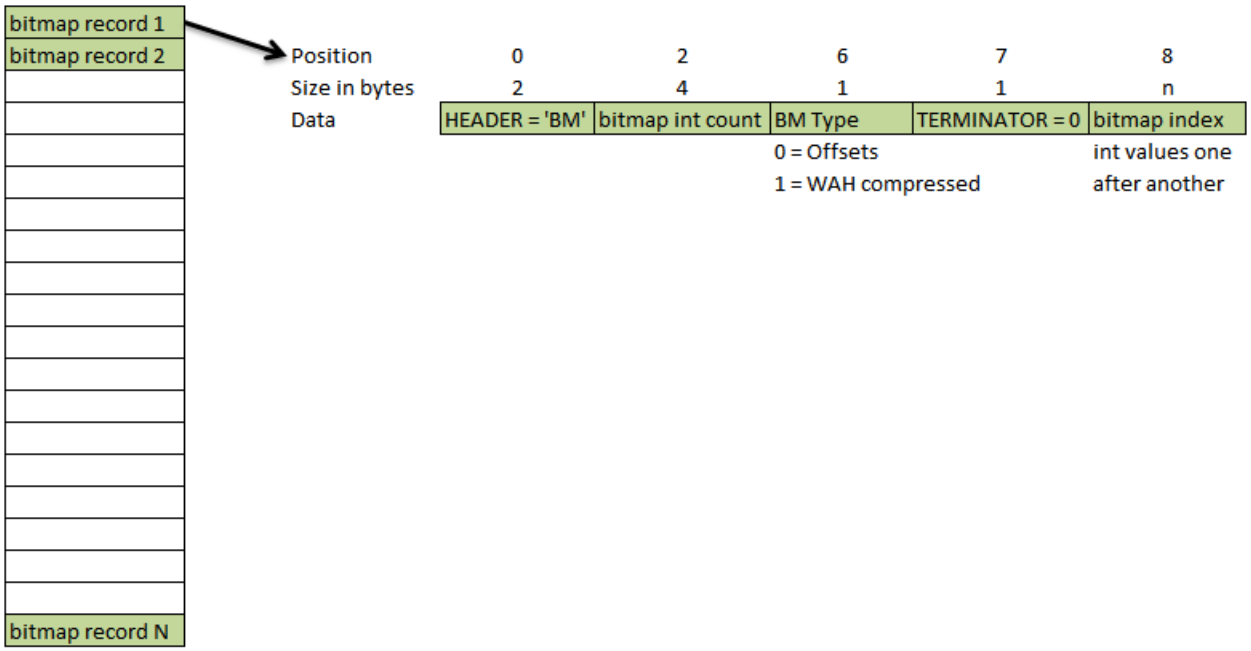
File Format : *.mgdat

Values are stored in the following structure on disk:



File Format : *.mgbmp

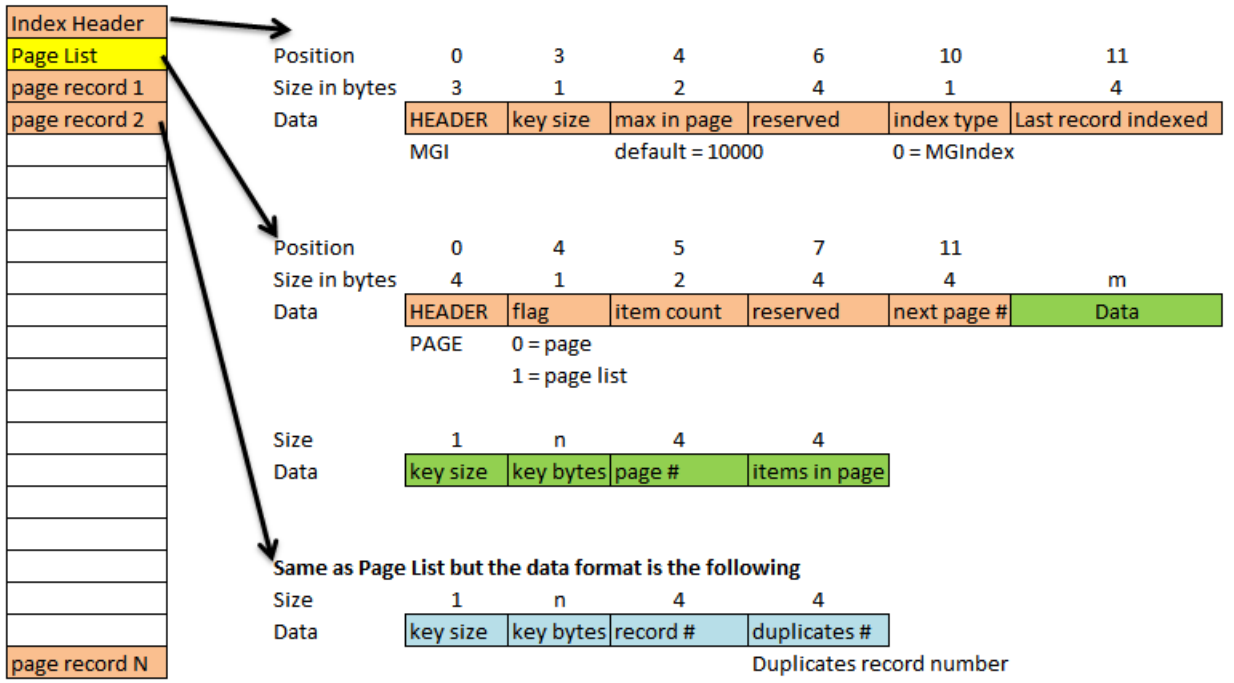
Bitmap indexes are stored in the following format on disk :



The bitmap row is variable in length and will be reused if the new data fits in the record size on disk, if not another record will be created. For this reason a periodic index compaction might be needed to remove unused records left from previous updates.

File Format : *.mgidx

The MGIndex index is saved in the following format as shown below:



File Format : *.mgbmr , *.mgrec

Rec file is a series of `long` values written to disk with no special formatting. These values map the record number to an offset in the BITMAP index file and DOCS storage file.

History

- **Initial Release v2.0** : 19th January 2012
- **Update v2.1** : 26th January 2012

  - lock on safedictionary iterator set, Thanks to **igalk474**
  - string default(T) -> "" instead of null, Thanks to **Ole Thrane** for finding it
  - mgindex string firstkey null fix
  - added test for normal string keys
  - fixed the link to the v1 article

- **Update v2.2** : 8th February 2012
  - bug fix removekey, Thanks to **syro_pro**
  - removed un-needed initialization in safedictionary, Thanks to **Paulo Zemek**

- **Update v2.3** : 1st March 2012
  - changed internal classes to structs (2x+ speed, 30% less memory)
  - added keystore class and code refactoring
  - added a v2.3 performance section to the article

- **Update v2.4** : 7th March 2012
  - bug fix remove key set page isDirty -> Thanks to **Martin van der Geer**
  - Page<T> is a class again to fix keeping it's state
  - added RemoveKeyTest unit test
  - removed MemoryStream from StorageFile.CreateRowHeader for speed
  - current record number is also set in the bitmap index for duplicates

- **Update v2.5** : 28th May 2012
  - added SafeSortedList for access concurrency of the page list
  - insert performance back to v2.3 speed (removed extra writing to duplicates)

- **Update v2.6** : 20th December 2012
  - post back code from RaptorDB the doc store
  - added more data types (uint,short,double,float,datetime...)
  - added locks to the indexfile
  - updated logger
  - updated safe dictionary with locks
  - changed to Path.DirectorySeparatorChar for Mono/MonoDroid support
  - bug fix edge case in WAHbitarray
  - updated storage file

- **Update v2.7.0** : 6th October 2013
  - bug fix WAHBitArray
  - index files are opened in shared mode for the ability of online copy backup
  - dirty pages are sorted on save for read performance

- **Update v2.7.5** : 11th October 2013
  - bug fix saving page list to disk for counts > 50 million items

License

About the Author

Mehdi Gholam

Architect -
United Kingdom

Mehdi first started programming when he was 8 on BBC+128k machine in 6512 processor language, after various hardware and software changes he eventually came across .net and c# which he has been using since v1.0.
He is formally educated as a system analyst Industrial engineer, but his programming passion continues.

* Mehdi is the 5th person to get 6 out of 7 Platinum's on Code-Project (13th Jan'12)
* Mehdi is the 3rd person to get 7 out of 7 Platinum's on Code-Project (26th Aug'16)

Comments and Discussions

325 messages have been posted for this article Visit **https://www.codeproject.com/Articles/316816/RaptorDB-The-Key-Value-Store-V2** to post and view comments on this article, or click **here** to get a print view with messages.