Binary Storage Solution Description

# Important Notes

* This solution requires reference to **System.Runtime.Caching.dll**, which is a .NET Framework DLL.
* Among all the requirements, only requirement **#101** is not implemented.
* Index table is implemented as **ConcurrentDictionary** and would be serialized to disk when performing **Add** operation. (1)
* As adding data to the Binary Storage should be transactional and if the Binary Storage is in the process of persisting data for a given key, the read operation should wait until persisting process complete, **ReaderWriterLockSlim** lock is implemented. Both **Add** and **Get** operation would acquire the same lock. However, **Add** operation would have higher-priority **Write** mode, while **Get** operation would have the lower-priority **Read** mode. (2)
* For caching storage stream on memory, **System.Runtime.Caching.MemoryCache** is used. It allows more flexibility on configuration. Compressed streams are cached to save memory. (3)
* To implement transaction during **Add**, a backup for storage file would be created before adding. In case of error, this backup file would override the current storage file. After that, added key is removed from index table and the table would then be reserialized back to disk to finish the rollback. (4)
* Some benchmarks:
  + 1000 files of 2.09 GB
    - Time to create: 00:01:29.5887978
    - Time to verify: 00:00:18.6763234

=> A **2MB** file took around **0.09s** to **Add** and **0.018s** to **Get**

* + 1000 files of 274 KB
    - Time to create: 00:00:36.5276995
    - Time to verify: 00:00:08.3162327

=> A **274B** file took around **0.036s** to **Add** and **0.008s** to **Get**

# Implementation details

| Id | Description |
| --- | --- |
| #1 | The solution should be implemented in C# language using .NET 4 or higher in Visual Studio 2015 or higher. Use of 3rd party libraries should be minimal and limited to open source public libraries. All sources should be provided. Solution should be standalone and should not require any additional software installed on the host machine like MS SQL Server, MS Message Queue, etc.  **Status**: **<DONE>** |
| #2.1 | *IBinaryStorage* interface should be implemented in a class called *BinaryStorage.* See attached Visual Studio solution.  **Status**: **<DONE>** |
| #2.2 | In the attached solution, *TestApp* project should not be modified. It will be used to test the code against the large set of files.  **Status**: **<DONE>** |
| #3 | The solution can use as much memory as the host system provides. But it should work on the systems with 1Gb of RAM and should not fail with OutOfMemory exceptions.  **Status**: **<DONE> Note**: BinaryStorage.ConfigureStorageCache method is used to set memory cache size limit to 1 GB. |
| #4 | It should be possible to configure the maximum amount of hard disk space that can be allocated for Storage File.  **Status**: **<DONE> Note**: BinaryStorage.CheckMaxStorageFile method verifies StorageConfiguration.MaxStorageFile setting before adding storage. |
| #5 | Index structure should have compact representation on the disk and in the memory. It should support arbitrary amount of records and might keep frequently used records in the memory to achieve the best performance during lookups for information.  **Status**: **<DONE> Note**: As mentioned in (1) |
| #6 | It should be possible to add a new data to the Binary Storage by providing a string key, an instance that implements stream and optional storage parameters (see *IBinaryStorage*).  Optional storage parameters:   * Hash of the data (MD5). If provided implementation should hash input stream while saving and verify hash match. If hashes don’t match the data should not be added to binary storage and exception should be thrown. * Compression flag. If this flag is present, the data is already compressed and therefore there is no benefit of compressing it again. * Length. If present indicates the length of the input stream.   When storage file is full or disk is full a proper exception should be thrown during attempt to add new data. If data with the same key is already present in the storage a proper exception should be thrown.  **Status**: **<DONE>** |
| #7 | Adding data to the Binary Storage should be transactional. If any error occurs during the addition, changes should be rolled back (records removed form Index, etc).  After Add method returns, the data should be saved to the disk and ready for consumption.  **Status**: **<DONE> Note**: As mentioned in (4) |
| #8 | It should be possible to retrieve binary data as stream from the Binary Storage by providing a key. The data retrieved from the storage should be exactly the same as the data previously added to the storage for persistence.  If stream was compressed during persistence (see also requirement #102) it should be transparently decompressed prior to sending it to the client.  It there is no data for a given key a proper exception should be thrown.  If the Binary Storage is in the process of persisting data for a given key, the read operation should wait until persisting process complete and then return data on success or exception otherwise.  **Status**: **<DONE> Note**: As mentioned in (2) |
| #9 | The Binary Storage should cache frequently accessed data in the main memory to minimize the number of read requests to hard drive.  **Status**: **<DONE> Note**: As mentioned in (3) |
| #10 | Adding and reading data from the Binary Storage should be thread-safe and multi-threaded. Thread-safe means that many threads might try to add data to the storage at the same time. Multi-threaded means that adding data distributed among 2 or more threads should be generally faster than adding the same data using only one thread.  **Status**: **<DONE> Note**: As mentioned in (2) |
| #11 | It should be possible to check if a given key is present in the Index  **Status**: **<DONE>** |

# Optional Requirements

| Id | Description |
| --- | --- |
| #101 | Duplicate binary data should be stored only once in the Storage File. Hash value should be used to check for duplicates.  **Status**: **<NOT DONE>** |
| #102 | Storage should compress an input stream during persistence (if it is not already compressed, see #6) to save disk space. However it might not be reasonable to compress a small stream. It should be possible to configure compression based on stream length (compress all streams with length greater than specified threshold).  It might be reasonable to store hashes of compressed and raw data to support de-duplication (#101).  **Status**: **<DONE> Note:** ObjectSerializer.Decompress method causes **OutOfMemory** error for large storage streams when using with MemoryStream, thus FileStream is implemented instead, along with temporary files. These temporary files are cleared frequently to free up disk space. |
| #103 | It should be possible to configure the maximum amount of hard disk space that can be allocated for storing Index.  **Status**: **<DONE> Note:** BinaryStorage.CheckMaxIndexFile method verifies StorageConfiguration.MaxIndexFile setting before adding storage. |