# Data Storage with PersistedDictionary

PersistedDictionary is a persisted data structure built on top of the ESENT database engine. It is a drop-in compatible replacement for the generic Dictionary<TKey,TValue> and SortedDictionary<TKey, TValue> classes.

# Quick Start

To get started with PersistedDictionary you should:

1. Copy Esent.Interop.dll and Esent.Collections.dll to your project.
2. Add a reference to Esent.Collections.dll.
3. Use the namespace ‘Microsoft.Isam.Esent.Collections.Generic’.
4. Pick a directory to contain the persisted dictionary. The directory will contain all the files (database, logfiles, and checkpoint) needed for the database that stores the data.
5. Create a PersistedDictionary, passing the directory name into the constructor.
6. Use the PersistedDictionary like an ordinary Dictionary or SortedDictionary.
7. Dispose of the PersistedDictionary when finished with it. If the dictionary isn’t disposed then ESENT will have the database open until the instance is finalized.

# Sample Code

Here is an application that remembers a first name → last name mapping in a persisted dictionary.

using System;

using Microsoft.Isam.Esent.Collections.Generic;

namespace PersistentDictionarySample

{

public static class HelloWorld

{

public static void Main()

{

var dictionary =

new PersistentDictionary<string, string>("Names");

Console.WriteLine("What is your first name?");

string firstName = Console.ReadLine();

if (dictionary.ContainsKey(firstName))

{

Console.WriteLine("Welcome back {0} {1}",

firstName,

dictionary[firstName]);

}

else

{

Console.WriteLine(

"I don't know you, {0}. What is your last name?",

firstName);

dictionary[firstName] = Console.ReadLine();

}

}

}

}

# Supported Key Types

Only these types are supported as dictionary keys:

|  |  |  |
| --- | --- | --- |
| Boolean | Byte | Int16 |
| UInt16 | Int32 | UInt32 |
| Int64 | UInt64 | Float |
| Double | Guid | DateTime |
| TimeSpan | String |  |

# Supported Value Types

Dictionary values can be any of the key types, Nullable<T> versions of the key types, Uri, IPAddress or a serializable structure. A structure is only considered serializable if it meets *all* these criteria:

* The structure is marked as serializable
* Every member of the struct is either:
  1. A primitive data type (e.g. Int32)
  2. A String, Uri or IPAddress
  3. A serializable structure.

Or, to put it another way, a serializable structure cannot contain any references to a class object. This is done to preserve API consistency. Adding an object to a PersistentDictionary creates a copy of the object though serialization. Modifying the original object will not modify the copy, which would lead to confusing behavior. To avoid those problems the PersistentDictionary will only accept value types as values.

|  |  |
| --- | --- |
| Can Be Serialized | Can’t Be Serialized |
| [Serializable]  struct Good  {  public DateTime? Received;  public string Name;  public Decimal Price;  public Uri Url;  } | [Serializable]  struct Bad  {  public byte[] Data; *// arrays aren’t supported*  public Exception Error; *// reference object*  } |

# Data Consistency

Each database update is performed in a separate transaction and the logfile and database updates are performed in the background. This means that dictionary updates are atomic and consistent, but not always durable. If the application crashes only updates whose log records have been written to disk will be recovered. It is possible to force the dictionary updates created so far to be persisted to disk using PersistedDictionary.Flush(). Note that every Flush() call requires a disk I/O so using this too frequently will severely limit the update rate. Disposing the dictionary will flush all its changes to disk.

# Performance Measurements

These are very basic performance measurements made with a PersistedDictionary<long, string> where the string data was 64 bytes in length. These measurements were taken on my laptop system.

|  |  |
| --- | --- |
| Sequential inserts | 12,000 entries/second |
| Random inserts | 3,500 entries/second |
| Random Updates | 10,000 entries/second |
| Random lookups (database cached in memory) | 30,000 entries/second |

Performance will vary from application to application. Factors that affect performance include:

* Total number of items in the dictionary.
* Size of the data. Retrieving an int will be faster than retrieving a 55MB string.
* How much of the database is cached in memory. When first opening a populated dictionary the data will not be cached in memory so initial lookups will be slow.
* Update patterns. Sequential inserts are much faster than random inserts.
* Structure serialization. Using structures as dictionary data can be much slower than using basic data types.
* Disk performance.

# File Management

The PersistedDictionary code will automatically open or create a database in the specified directory. To determine if a dictionary database already exists use PersistentDictionaryFile.Exists(). To remove a dictionary use PersistentDictionaryFile.DeleteFiles().

# PersistedDictionary Features

* **No setup**: the ESENT database engine is part of Windows and requires no setup. EsentCollections will work on any version of Windows from XP onwards.
* **Performance**: ESENT supports a high rate of updates and retrieves. Write-ahead logging decreases the cost of making small updates to the data. Information is inserted into or retrieved from the database locally so data access has very low overhead.
* **Simplicity**: a PersistedDictionary looks and behaves like the .NET Dictionary and SortedDictionary classes. No extra method calls are needed.
* **Administration-free**: ESENT automatically manages the database cache size, transaction logfiles, and crash recovery so no database administration is needed. The code is structured so that there are no deadlocks or conflicts, even when multiple threads use the same dictionary. ESENT runs in-process and doesn’t expose any network access, providing a high degree of security.
* **Reliability:** ESENT’s write-ahead logging system means that a database can be recovered after a process crash or unexpected machine shutdown (e.g. power outage). Database transactions are used to ensure the logical consistency of the database.
* **Concurrency**: each data structure can be accessed by multiple threads. Reads are non-blocking and updates to different items in the collection are allowed to proceed concurrently.
* **Scale**: A collection can contain up to 2^31 objects[[1]](#footnote-1), and string values can be up to 2GB in size. The maximum database size is 16TB[[2]](#footnote-2).

1. Mostly limited by the fact that the Count property on the generic collection objects returns an integer. ESENT has no restriction on the number of records in a table. [↑](#footnote-ref-1)
2. Typical ESENT database sizes range from a few MB (small client applications) to a few TB (server applications). [↑](#footnote-ref-2)