» Database » Database » Utilities

A Fast CSV Reader

By Sebastien Lorion | 6 Jul 2011

Licence	MIT
First Posted	9 Jan 2005
Views	2,327,565
Downloads	31,112
Bookmarked	1,007 times

.NET1.1 .NET2.0 VS.NET2003 VS2005 C# Windows DBA Dev Intermediate

A reader that provides fast, non-cached, forward-only access to CSV data.

★★★★★ 4.92 (374 votes)

Download source files for .NET 2.0 - 539 KB
Download binaries for .NET 2.0 - 23.7 KB

Introduction

One would imagine that parsing CSV files is a straightforward and boring task. I was thinking that too, until I had to parse several CSV files of a couple GB each. After trying to use the OLEDB JET driver and various Regular Expressions, I still ran into serious performance problems. At this point, I decided I would try the custom class option. I scoured the net for existing code, but finding a *correct*, *fast*, and *efficient* CSV parser and reader is not so simple, whatever platform/language you fancy.

I say correct in the sense that many implementations merely use some splitting method like `String.Split()`. This will, obviously, not handle field values with commas. Better implementations may care about escaped quotes, trimming spaces before and after fields, etc., but none I found were doing it all, and more importantly, in a **fast** and **efficient** manner.

And, this led to the CSV reader class I present in this article. Its design is based on the `System.IO.StreamReader` class, and so is a non-cached, forward-only reader (similar to what is sometimes called a fire-hose cursor).

Benchmarking it against both OLEDB and regex methods, it performs about 15 times faster, and yet its memory usage is very low.

To give more down-to-earth numbers, with a 45 MB CSV file containing 145 fields and 50,000 records, the reader was processing about 30 MB/sec. So all in all, it took 1.5 seconds! The machine specs were P4 3.0 GHz, 1024 MB.

- Read about the latest updates here

Supported Features

This reader supports fields spanning multiple lines. The only restriction is that they must be quoted, otherwise it would not be possible to distinguish between malformed data and multi-line values.

Basic data-binding is possible via the `System.Data.IDataReader` interface implemented by the reader.

You can specify custom values for these parameters:

- Default missing field action;
- Default malformed CSV action;

- Buffer size;
- Field headers option;
- Trimming spaces option;
- Field delimiter character;
- Quote character;
- Escape character (can be the same as the quote character);
- Commented line character.

If the CSV contains field headers, they can be used to access a specific field.

When the CSV data appears to be malformed, the reader will fail fast and throw a meaningful exception stating where the error occurred and providing the current content of the buffer.

A cache of the field values is kept for the current record only, but if you need dynamic access, I also included a cached version of the reader, `CachedCsvReader`, which internally stores records as they are read from the stream. Of course, using a cache this way makes the memory requirements way higher, as the full set of data is held in memory.

Latest Updates (3.8 Release)

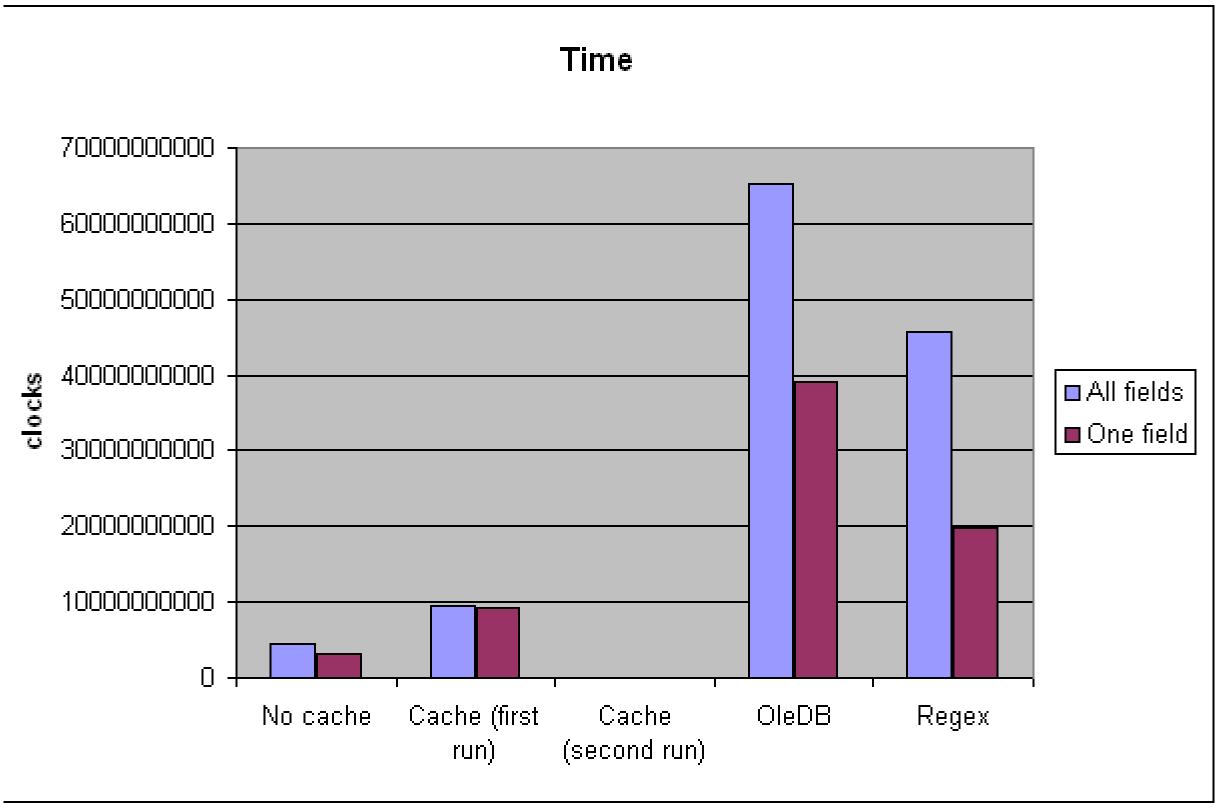
- Empty header names in csv files are now replaced by a default name that can be customized via the new DefaultHeaderName property (by default, it is "Column" + column index).

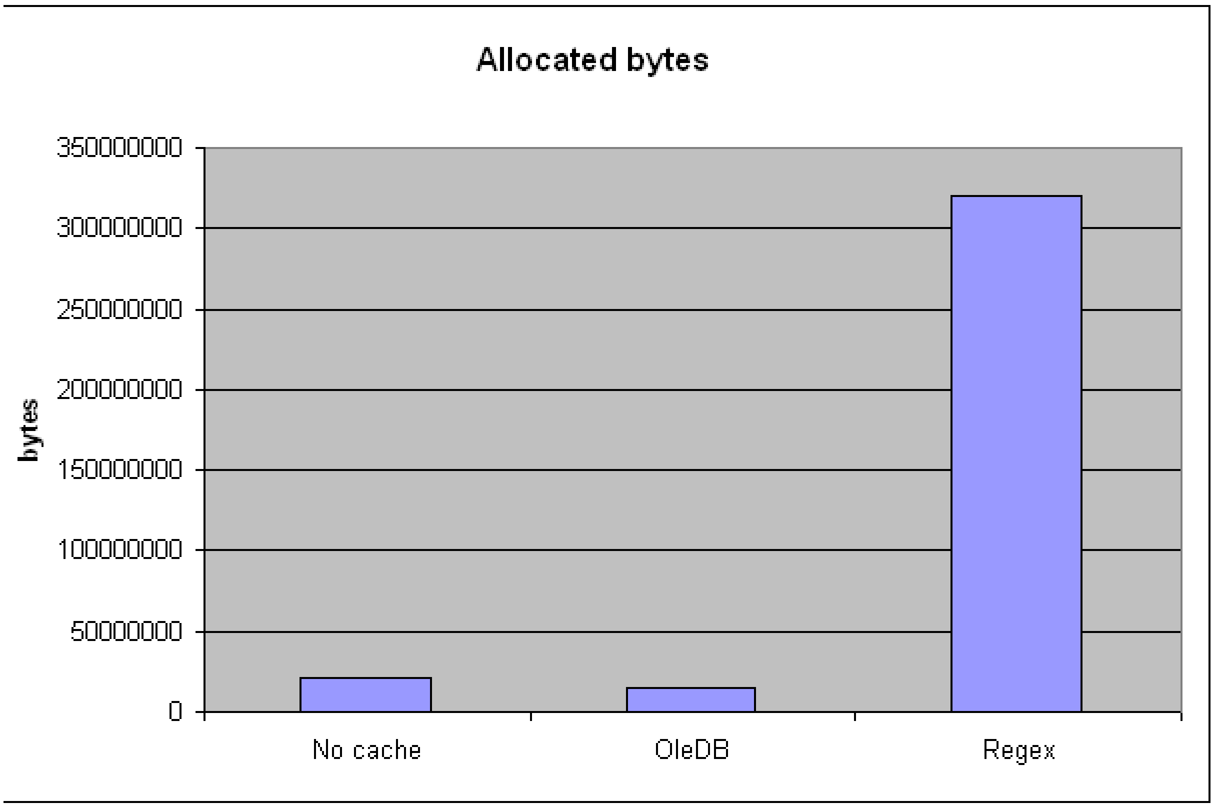
Benchmark and Profiling

You can find the code for these benchmarks in the demo project. I tried to be fair and follow the same pattern for each parsing method. The regex used comes from Jeffrey Friedl's book, and can be found at page 271. It doesn't handle trimming and multi-line fields.

The test file contains 145 fields, and is about 45 MB (included in the demo project as a RAR archive).

I also included the raw data from the benchmark program and from the CLR Profiler for .NET 2.0.





Using the Code

The class design follows `System.IO.StreamReader` as much as possible. The parsing mechanism introduced in version 2.0 is a bit trickier because we handle the buffering and the new line parsing ourselves. Nonetheless, because the task logic is clearly encapsulated, the flow is easier to understand. All the code is well documented and structured, but if you have any questions, simply post a comment.

Basic Usage Scenario

```
using System.IO;
using LumenWorks.Framework.IO.Csv;

void ReadCsv()
{
    // open the file "data.csv" which is a CSV file with headers
    using (CsvReader csv =
           new CsvReader(new StreamReader("data.csv"), true))
    {
        int fieldCount = csv.FieldCount;
        string[] headers = csv.GetFieldHeaders();

        while (csv.ReadNextRecord())
        {
            for (int i = 0; i < fieldCount; i++)
                Console.Write(string.Format("{0} = {1};",
                              headers[i], csv[i]));

            Console.WriteLine();
        }
    }
}
```

Simple Data-Binding Scenario (ASP.NET)

```
using System.IO;
using LumenWorks.Framework.IO.Csv;

void ReadCsv()
{
    // open the file "data.csv" which is a CSV file with headers
    using (CsvReader csv = new CsvReader(
                           new StreamReader("data.csv"), true))
    {
        myDataRepeater.DataSource = csv;
        myDataRepeater.DataBind();
    }
}
```

Complex Data-Binding Scenario (ASP.NET)

Due to the way both the `System.Web.UI.WebControls.DataGrid` and `System.Web.UI.WebControls.GridView` handle `System.ComponentModel.ITypedList`, complex binding in ASP.NET is **not possible**. The only way around this limitation would be to wrap each field in a container implementing `System.ComponentModel.ICustomTypeDescriptor`.

Anyway, even if it was possible, using the simple data-binding method is much more efficient.

For the curious amongst you, the bug comes from the fact that the two grid controls completely ignore the property descriptors returned by `System.ComponentModel.ITypedList`, and relies instead on `System.ComponentModel.TypeDescriptor.GetProperties(...)`, which obviously returns the properties of the string array and not our custom properties. See `System.Web.UI.WebControls.BoundColumn.OnDataBindColumn(...)` in a disassembler.

Complex Data-Binding Scenario (Windows Forms)

```
using System.IO;
using LumenWorks.Framework.IO.Csv;

void ReadCsv()
{
    // open the file "data.csv" which is a CSV file with headers
    using (CachedCsvReader csv = new
           CachedCsvReader(new StreamReader("data.csv"), true))
    {
        // Field headers will automatically be used as column names
        myDataGrid.DataSource = csv;
    }
}
```

Custom Error Handling Scenario

```
using System.IO;
using LumenWorks.Framework.IO.Csv;

void ReadCsv()
{
    // open the file "data.csv" which is a CSV file with headers
    using (CsvReader csv = new CsvReader(
           new StreamReader("data.csv"), true))
    {
        // missing fields will not throw an exception,
        // but will instead be treated as if there was a null value
        csv.MissingFieldAction = MissingFieldAction.ReplaceByNull;

        // to replace by "" instead, then use the following action:
        //csv.MissingFieldAction = MissingFieldAction.ReplaceByEmpty;

        int fieldCount = csv.FieldCount;
        string[] headers = csv.GetFieldHeaders();

        while (csv.ReadNextRecord())
        {
            for (int i = 0; i < fieldCount; i++)
                Console.Write(string.Format("{0} = {1};",
                              headers[i],
                              csv[i] == null ? "MISSING" : csv[i]));

            Console.WriteLine();
        }
    }
}
```

Custom Error Handling Using Events Scenario

```
using System.IO;
using LumenWorks.Framework.IO.Csv;

void ReadCsv()
{
    // open the file "data.csv" which is a CSV file with headers
    using (CsvReader csv = new CsvReader(
           new StreamReader("data.csv"), true))
    {
        // missing fields will not throw an exception,
        // but will instead be treated as if there was a null value
        csv.DefaultParseErrorAction = ParseErrorAction.RaiseEvent;
        csv.ParseError += new ParseErrorEventHandler(csv_ParseError);

        int fieldCount = csv.FieldCount;
        string[] headers = csv.GetFieldHeaders();

        while (csv.ReadNextRecord())
        {
                for (int i = 0; i < fieldCount; i++)
                    Console.Write(string.Format("{0} = {1};",
                                  headers[i], csv[i]));

                Console.WriteLine();
            }
        }
}

void csv_ParseError(object sender, ParseErrorEventArgs e)
{
    // if the error is that a field is missing, then skip to next line
    if (e.Error is MissingFieldCsvException)
    {
        Console.Write("--MISSING FIELD ERROR OCCURRED");
        e.Action = ParseErrorAction.AdvanceToNextLine;
    }
}
```

History

Version 3.8 (2011-07-05)

- Empty header names in CSV files are now replaced by a default name that can be customized via the new `DefaultHeaderName` property (by default, it is "Column" + column index).

Version 3.7.2 (2011-05-17)

- Fixed a bug when handling missing fields.
- Strongly named the main assembly.

Version 3.7.1 (2010-11-03)

- Fixed a bug when handling whitespaces at the end of a file.

Version 3.7 (2010-03-30)

- **Breaking**: Added more field value trimming options.

Version 3.6.2 (2008-10-09)

- Fixed a bug when calling `MoveTo` in a particular action sequence;
- Fixed a bug when extra fields are present in a multiline record;
- Fixed a bug when there is a parse error while initializing.

Version 3.6.1 (2008-07-16)

- Fixed a bug with `RecordEnumerator` caused by reusing the same array over each iteration.

Version 3.6 (2008-07-09)

- Added a web demo project;
- Fixed a bug when loading `CachedCsvReader` into a `DataTable` and the CSV has no header.

Version 3.5 (2007-11-28)

- Fixed a bug when initializing `CachedCsvReader` without having read a record first.

Version 3.4 (2007-10-23)

- Fixed a bug with the `IDataRecord` implementation where `GetValue`/`GetValues` should return `DBNull.Value` when the field value is empty or `null`;
- Fixed a bug where no exception is raised if a delimiter is not present after a non final quoted field;
- Fixed a bug when trimming unquoted fields and whitespaces span over two buffers.

Version 3.3 (2007-01-14)

- Added the option to turn off skipping empty lines via the property `SkipEmptyLines` (on by default);
- Fixed a bug with the handling of a delimiter at the end of a record preceded by a quoted field.

Version 3.2 (2006-12-11)

- Slightly modified the way missing fields are handled;
- Fixed a bug where the call to `CsvReader.ReadNextRecord()` would return `false` for a CSV file containing only one line ending with a new line character and no header.

Version 3.1.2 (2006-08-06)

- Updated dispose pattern;
- Fixed a bug when `SupportsMultiline` is `false`;
- Fixed a bug where the `IDataReader` schema column "DataType" returned `DbType.String` instead of `typeof(string)`.

Version 3.1.1 (2006-07-25)

- Added a `SupportsMultiline` property to help boost performance when multi-line support is not needed;
- Added two new constructors to support common scenarios;
- Added support for when the base stream returns a length of 0;
- Fixed a bug when the `FieldCount` property is accessed before having read any record;
- Fixed a bug when the delimiter is a whitespace;
- Fixed a bug in `ReadNextRecord(...)` by eliminating its recursive behavior when initializing headers;
- Fixed a bug when EOF is reached when reading the first record;
- Fixed a bug where no exception would be thrown if the reader has reached EOF and a field is missing.

Version 3.0 (2006-05-15)

- Introduced equal support for .NET 1.1 and .NET 2.0;
- Added extensive support for malformed CSV files;
- Added complete support for data-binding;
- Made available the current raw data;
- Field headers are now accessed via an array (breaking change);
- Made field headers case insensitive (thanks to Marco Dissel for the suggestion);
- Relaxed restrictions when the reader has been disposed;
- `CsvReader` supports 2^63 records;
- Added more test coverage;
- Upgraded to .NET 2.0 release version;
- Fixed an issue when accessing certain properties without having read any data (notably `FieldHeader`s).

Version 2.0 (2005-08-10)

- Ported code to .NET 2.0 (July 2005 CTP);
- Thoroughly debugged via extensive unit testing (special thanks to shriop);
- Improved speed (now 15 times faster than OLEDB);
- Consumes half the memory than version 1.0;
- Can specify a custom buffer size;
- Full Unicode support;
- Auto-detects line ending, be it *\r*, *\n*, or *\r\n*;
- Better exception handling;
- Supports the *"field1\rfield2\rfield3\n"* pattern (used by Unix);
- Parsing code completely refactored, resulting in much cleaner code.

Version 1.1 (2005-01-15)

- 1.1: Added support for multi-line fields.

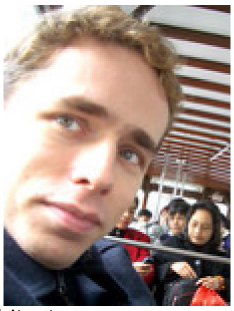
Version 1.0 (2005-01-09)

- 1.0: First release.

License

About the Author

Sebastien Lorion



Architect

Canada

Member

Sébastien Lorion is software architect as day job.

He is also a musician, actually singing outside the shower

He needs constant mental and emotional stimulation, so all of this might change someday ...

You can visit his blog at http://sebastienlorion.com.

Comments and Discussions

1937 messages have been posted for this article Visit **http://www.codeproject.com/KB/database/CsvReader.aspx** to post and view comments on this article, or click **here** to get a print view with messages.