crystal decisions™

Crystal Reports for Visual Studio .NET

Scaling Crystal Reports for Visual Studio .NET

Table of Contents

Best Practices

In general, there are many best practices for designing applications for scalability. The following sections provide some scalability best practices for applications using Crystal Reports for Visual Studio .NET in WebForms or Web Services scenarios.

Report Performance

An application will only be as fast as the reports it uses. A large part of tuning an application for scalability is ensuring that all the reports are designed in such a way that they will perform at an optimum speed

Report size is one of the obvious factors in report performance. When designing for scalability consider using smaller reports, generally less then 15 pages. Often with web-based applications, users simply want to see a few key pieces of data anyway, rather than be presented with pages and pages of data.

The same reports that you may have used in the past in desktop applications may not be appropriate for web-based applications. To make reports smaller, use the record selection formula to limit the number of records returned to the report. The record selection formula can be hard coded in the report, or it may be set at runtime via the SelectionFormula property of the viewers, or via the RecordSelectionFormula property of the ReportDocument. Parameters can also be used to make the report's records more dynamic and focused.

Subreports

While subreports can often be the used to create complex reports, they have a fair amount of overhead associated with them. This doesn't mean that subreports shouldn't be used, but it does mean that they should be used sparingly.

Also consider that when subreports are placed in repeating sections such as group headers or the details section, each instance of that subreport requires it's own report job. This means that a 100-record report with a subreport in the details section will actually be running 101 report jobs.

On-Demand Processing

Crystal Reports does process reports on-demand. For example, if the first page of a report is to be displayed, it will only process that first page, not the whole report.

However, inserting fields or formulas onto the first page of a report that depend on the completion of the report processing will prevent the on-demand processing from happening. An example of this is a "Page N of M" special field.

If this special field is placed on the first page, the entire report must be processed in order to determine the page count before this field can be rendered. This would cause the first page to have to wait until the whole report was finished processing before it could be displayed. The same applies to percentage of summary calculations. If performance and scalability are a factor for your application, consider eliminating the use of these types of fields.

Graphics

One of the powerful features of Crystal Reports is its ability to display rich graphics on reports in the form of charts and images. Like subreports, they do not need to be eliminated from reports when scalability is a factor, but they should be used sparingly.

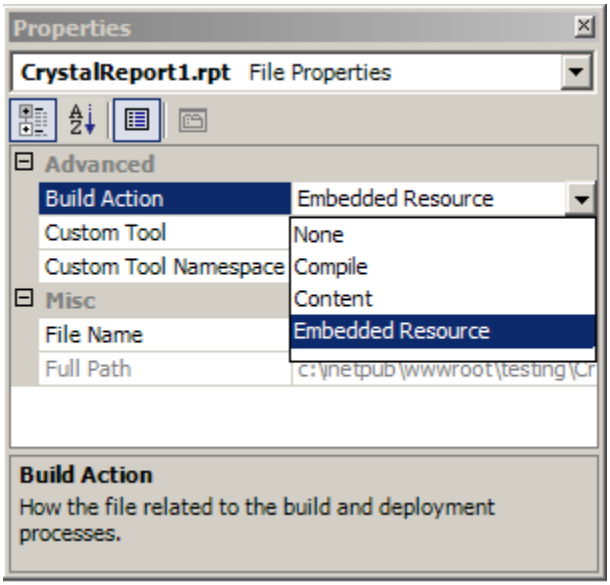
When displaying report using the Web Forms report viewer, for each chart or image on the report page, the report engine must create a temporary JPG file (in the dynamic_images folder under the CrystalReportWebFormViewer virtual directory), and the browser must make a separate request to the web server to retrieve it.

The Web Forms viewer also periodically searches through the dynamic_images folder for old images and deletes them. This disk access can reduce report performance, so using charts and images sparingly will keep the disk access to a minimum.

Loading the report

A final point to mention about report performance is where the report is stored. By default, when reports are added to a Visual Studio .NET project, they are compiled into the output assembly as an embedded resource. When referenced by classname a copy of the report is extracted from the assembly and written to a temp file, and then the report job is opened from the temp file. This extra step introduces unneeded overhead.

When performance is a factor, use the ReportDocument class directly, calling the Load method to load a report file from disk. Also, change the report to not be embedded in your assembly; this will make your assemblies smaller. To accomplish this, select the report in the solution explorer and change the build action property as seen in the screenshot below. This will save a few CPU cycles which when multiplied by many users makes a worthwhile performance impact.



DataSets and Performance

The ability to use an ADO.NET DataSet as the data source for a report is a powerful feature. It allows the developers to perform their own query against the database, or perhaps even construct a DataSet programmatically without a database.

For more information on this scenario, visit the Crystal Decisions' support site at http://support.crystaldecisions.com/docs and search for the ADO.NET Walk through technical brief:

Rtm_reportingoffadonetdatasets.pdf

Reporting from Datasets has it's advantages, but it shouldn't necessarily be used as an exclusive data access mechanism. If your application creates a DataSet for the sole purpose of the report, and especially if the query is simple such as *SELECT field1, field2 FROM table* then there is little value in using a DataSet.

Having Crystal Reports connect directly to the database and perform the query would be faster and require less memory. On the other hand, DataSets are useful when data from multiple data sources needs to be combined, or perhaps the data inside the DataSet needs to be massaged after the query is run. For the purpose of this paper, let's examine the scalability implications of reporting from DataSets.

The report engine internally attempts to perform a certain level of database connection pooling. That is, if 3 users hit the same ASPX page with the same report on it, the same database connection will most likely be used to perform all three queries.

One thing the report engine does not do is pooling of the query itself. For each report job (each ReportDocument instance), a database query must be performed. If the query takes a while to process, or if the set of data returned is very large, then performing this query on a per-user basis could turn out to be the bottleneck of the application.

Performing a per-user query into a DataSet would be even worse because of the memory overhead. However, one way that ADO.NET can be useful is to act as a cache for the query results. Instead of creating a DataSet on each hit to the aspx page, perform the query the first time, then store the DataSet in the ASP.NET cache, retrieving it from there later. This allows all report jobs to share a single DataSet and thus reduce the number of queries to the database. The C# code listing below illustrates how this could be accomplished. This logic could be ported to VB.NET.

```
DataSet ds;

// use the filename of the report as the cache lookup key
```

```
string datasetCacheKey = report.FilePath;

if (Cache[datasetCacheKey] != null)
{
   ds = (DataSet) Cache[datasetCacheKey];
}
else
{
   ds = new DataSet();

   // perform query and fill dataset
   ...

   // insert the dataset into the cache
   Cache.Insert(datasetCacheKey, ds);
}

// set the dataset as the data source for the first table in the
report
report.Database.Tables[0].SetDataSource(ds);
```

The first hit to this page will not be any faster than without using caching, but on subsequent hits, you should notice the response time being much quicker.

The last note on Datasets is that if more than one table are used in a report, the joins have to be performed inside the report engine instead of the database. It's better to run a query that does the joining on the database and return a single result set as a table.

Report Job Caching

In the previous section, the concept of caching DataSets was explained. This section will describe report job caching, which will produce an even larger performance improvement.

Each instance of a ReportDocument object represents a report job. Each report job holds in memory, among other things, the report definition, any state such as parameter values, database connections, and also the query result set whether that be based on a DataSet or a direct database connection. In most cases, each hit to an ASPX page will open a new report job.

After the request is completed, the ReportDocument object and the print job will be destroyed. This will happen even when the user clicks the Next Page button to show the following page of the same report.

An easy performance gain is to cache the report job, keeping it alive across requests. In this case of caching, we aren't actually storing data in the cache like you might expect, but rather just storing an instance of the ReportDocument in the cache. Keeping an instance of the ReportDocument alive keeps the print job alive, which results is less overhead preparing to display each page.

The C# code listing below (much like the previous code listing where a DataSet was being cached) illustrates a simple report job caching mechanism.

```
ReportDocument report;
// use the RPT filename as the cache lookup key
string reportCacheKey =
Server.MapPath("CrystalReport1.rpt");

if (Cache[reportCacheKey] != null)
{
   report = (ReportDocument) Cache[reportCacheKey];
}
else
{
   report = new ReportDocument();

   report.Load(Server.MapPath("CrystalReport1.rpt"));

   // perform any operations here such as passing
parameters
   // or providing database logon credentials

   Cache.Insert(reportCacheKey, report);
}

CrystalReportViewer1.ReportSource = report;
```

On the first hit to this page the report job will be opened. On subsequent hits to this page, the report job will be retrieved from the cache and the response time will be quicker.

As simple as this is, these few lines of code could produce big performance gains. However, this logic has some potential flaws. Because the report filename is used as the cache lookup key, there is no distinction between parameter values or logon credentials.

For example, if you had this page set up so it could accept requests like the following:

http://yourserver/reports/viewSalesReport.aspx?Country=USA

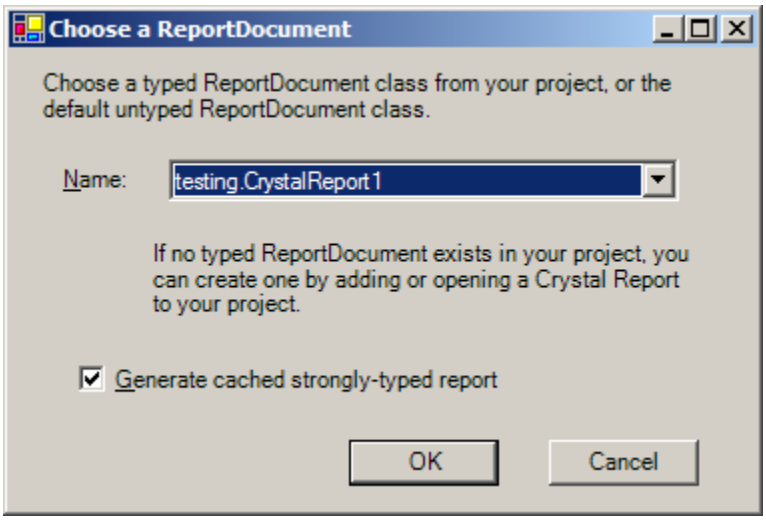
or

http://yourserver/reports/viewSalesReport.aspx?Country=Japan

After you loaded the report, you extracted the Country value from the query string and used it as a parameter value. Because the cache look up key doesn't not take the parameter value into account both users would see the same report, which means that for one of the users (whoever's request came second) would see incorrect data since it would be using the cached report job.

Ideally more items than just the report filename should be used in the cache key in order to distinguish between different requests. Luckily, you don't need to code this all yourself as this is a built-in feature inside Crystal Reports for Visual Studio .NET.

If you've ever dragged a ReportDocument from the toolbox onto a Web Form, you will have seen the following dialog pop up:



This dialog allows you to look up the instance of the ReportDocument you are creating with a report file already in your project. For the purposes of this paper, the important thing to notice is the checkbox near the bottom of the dialog. If this checkbox is on (it is on by default), then Crystal Reports will perform report job caching for you. If you uncheck it, then the report job will be re-opened on each hit to the page.

When you instruct Crystal Reports to perform the caching for you, it does something very similar to the previous code example, by storing the ReportDocument in the ASP.NET cache. However, it has a more robust implementation than the simple code listing above. It takes into account both parameter values and database logon credentials so that the two requests for the same report uses different parameter values and not share the same report job.

Turning the caching feature on via the checkbox in the user interface is fine if you are using the ReportDocument toolbox item, but if you are coding manually, you'll need to understand a bit more about how this caching mechanism works.

Caching mechanism

Crystal Reports determines whether to cache the report job if the object that is set into the ReportSource property of the report viewer, implements an interface called ICachedReport.

Found in the CrystalDecisions.ReportSource namespace of the CrystalDecisions.ReportSource.dll assembly, the ICachedReport interface tells Crystal Reports two things:

- It indicates that the report job should be cached
- It provides some information about how the caching should happen

By default, whenever you add a report file to your project, the code generation creates two classes in a code-behind file (click the Show All Files button in the solution explorer to be able to see this code-behind file). If your report file was named FooReport.rpt, you would get the following two classes:

```
public class FooReport : ReportClass

public class CachedFooReport : Component, ICachedReport
```

The FooReport class is derived from the ReportClass, which in turn derives from ReportDocument. If you set an instance of FooReport to the ReportSource property of the Web Forms viewer, Crystal Reports will perform no caching.

However, you set an instance of CachedFooReport to the ReportSource property, the fact that it implements the ICachedReport interface indicates that it should be cached. In fact, all the "Generate cached strongly-typed report" checkbox does is determine which type of class to create when you drop the ReportDocument object.

If you examine the implementation of this class, you'll find that the interface consists of 3 properties and 2 methods. Below is a description of each of these properties and methods:

`public virtual Boolean IsCacheable` - This property indicates whether the report should be cached. This should generally always return true.

`public virtual Boolean ShareDBLogonInfo` - This property indicates whether the database logon credentials should be treated as a differentiator for report jobs. This should return false unless you have row-level security applied to your database.

`public virtual TimeSpan CacheTimeOut` - This property indicated what the cache timeout should be for the report job. The default is 2 hours and 10 minutes. Any valid TimeSpan value can be returned here.

`public virtual ReportDocument CreateReport()` - This method is where Crystal Reports asks to be passed an instance of the ReportDocument to use. By default, this just returns an instance of the corresponding ReportDocument, a FooReport in our example. Any parameter values or other customization could be performed here.

`public virtual String GetCustomizedCacheKey(RequestContext request)` - This method usually returns null which means that Crystal Reports will manage the cache look up key itself, but this implementation can be changed to use your own logic. An example of this is appending on the current time to the cache key so that any requests for the report after a certain minute or hour will force a new job to be created.

In most cases, this code is not modified but you may create your own class that implements ICachedReport. Below is an example of such a class.

```
using System;
using CrystalDecisions.Shared;
using CrystalDecisions.ReportSource;
using CrystalDecisions.CrystalReports.Engine;

namespace MyWebApplication
{
   public class ReportCacher : ICachedReport
   {
      private string reportFile;
      private ReportDocument report;
      public ReportCacher(string reportFile)
      {
         this.reportFile = reportFile;
      }
      // ICachedReport implementation
      public virtual Boolean IsCacheable
      {
         get { return true; }
         set { }
      }
      public virtual Boolean ShareDBLogonInfo
      {
         get { return false;     }
         set { }
      }
      public virtual TimeSpan CacheTimeOut
      {
         get { return
CachedReportConstants.DEFAULT_TIMEOUT; }
         set { }
      }
      public virtual ReportDocument CreateReport()
      {
         if (report == null)
         {
            report = new ReportDocument();
            report.Load(reportFile);
```

```
            }
            return report;
        }
        public virtual String
GetCustomizedCacheKey(RequestContext request)
        {
            return null;
        }
    }
}
```

This class could be used like this:

```
string fileName = Server.MapPath("CrystalReport1.rpt")
ReportCacher cacher = new ReportCacher(fileName);

CrystalReportViewer1.ReportSource = cacher;
```

Exporting

With the wide array of export formats that Crystal Reports supports, many web applications are designed to return exported reports in formats such as PDF and Excel.

Since exporting is a processor-intensive operation when scalability is a factor, it's best to handle this process by implementing some mechanism to cache the output of exports so that the same file may be reused as a result for multiple requests. Writing the file to a temporary location on the disk but not deleting them most easily does this.

Use a lookup table of some kind to determine which request should return which report, and stream back that file from disk without re-processing it on each request.

Benchmarking Metrics

So far, we've discussed many best practices around designing an application for optimal performance under a heavy user load. While this is useful, it's also useful to have some benchmarking metrics to use as an estimation point when planning the application. The following section shares the results of some benchmarking that Crystal Decisions has completed.

Test scenario

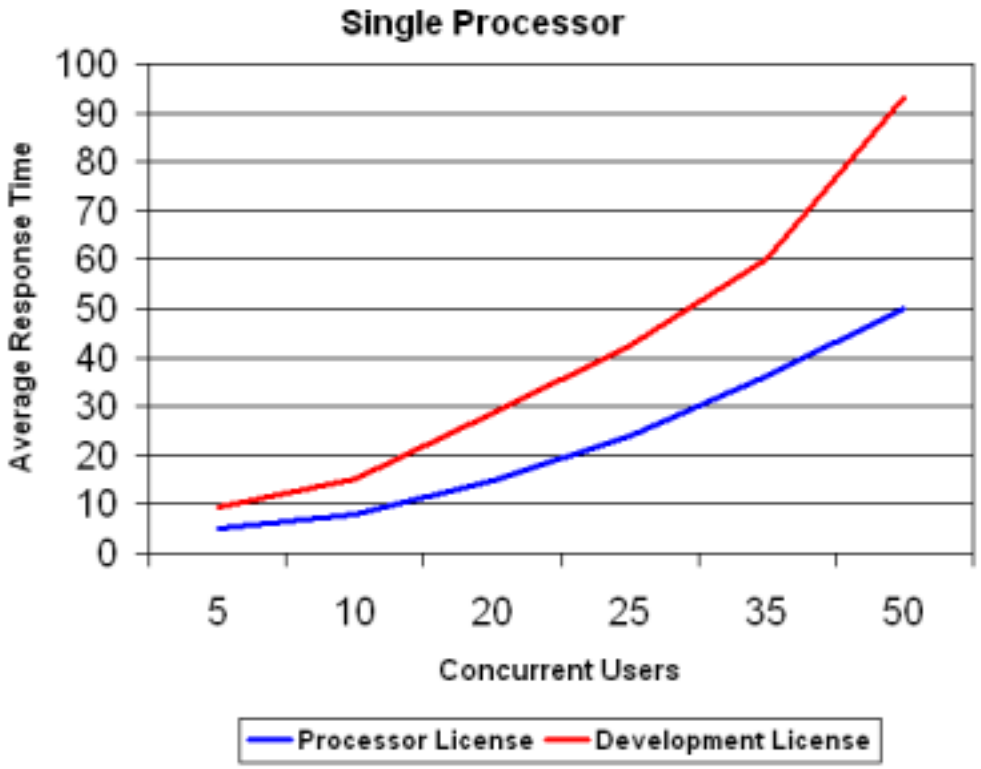
The machine used for the benchmarking was a Compaq Proliant 580 with 1.5GB of RAM. Scenarios with single and quad 700 Mhz CPU configurations were tested.

The reports used the TPC Benchmarking database held in SQL Server as a data source and connected via ODBC. The report used in the test was 168 pages long, was returning 1680 records and included multiple charts and images, as well as one-level of grouping. The report was considered moderately complex in nature and had no subreports.

Results

Single Processor

The following chart illustrates the average response time using a single processor configuration plotted against the number of concurrent users. The red line depicts the scalability of the report engine as included in the Visual Studio .NET box. The blue line depicts the scalability of the report engine with a Processor license applied.

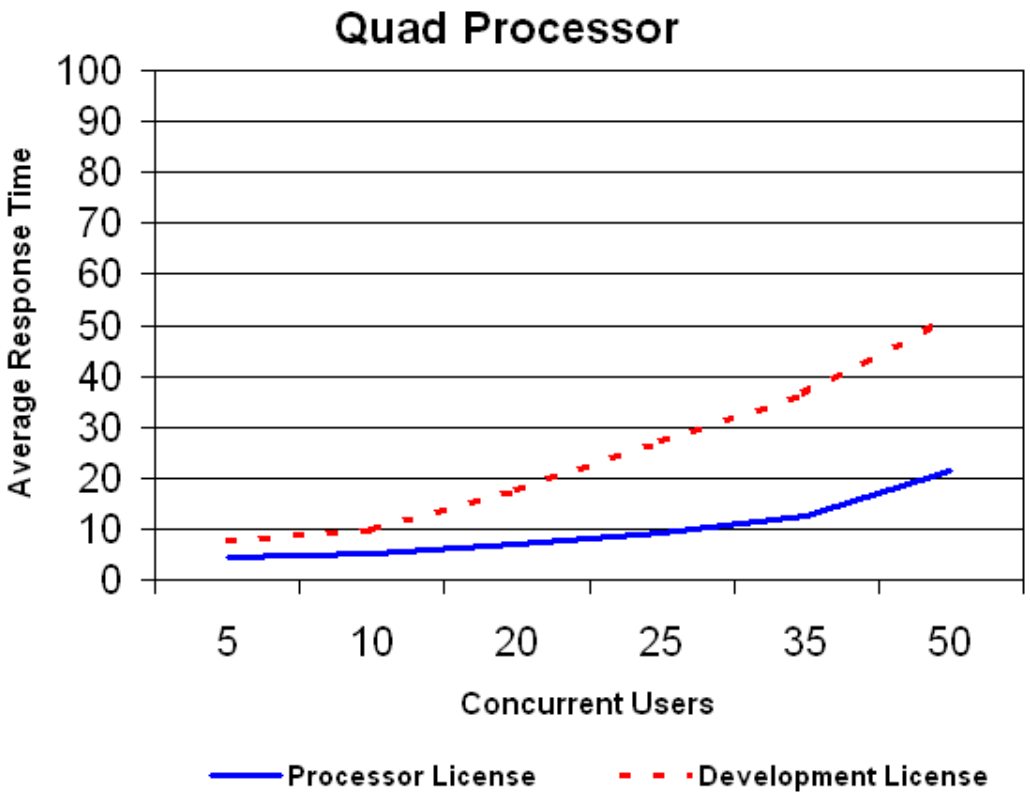


Note that with the processor license, up to 25 concurrent users requesting report pages can be served with a below-20 second response time.

Quad Processor

The following chart illustrates the average response time using a quad processor configuration plotted against the number of concurrent users.

The red line depicts the scalability of the report engine as included in the Visual Studio .NET box. The blue line depicts the scalability of the report engine with a Processor license applied.



Note that with the processor license on a quad CPU machine, up to 50 concurrent users requesting report pages can be served with a below-20 second response time. This is double the number than a single processor. Adding more CPUs either on the same machine would increase this in a greater fashion.

Neither of the above tests had any caching enabled therefore adding a good caching scheme could radically improve these numbers.

Contacting Crystal Decisions for Technical Support

We recommend that you refer to the product documentation and that you visit our Technical Support web site for more resources.

Self-serve Support:

http://support.crystaldecisions.com/

Email Support:

http://support.crystaldecisions.com/support/answers.asp

Telephone Support:

http://www.crystaldecisions.com/contact/support.asp