Crystal Reports 2008 Component Engine Scalability

Applies to:

Crystal Reports® 2008 .Net runtime engine, Crystal Reports version for Eclipse 2.0 Java runtime engine, Crystal Reports Server Embedded 2008, BusinessObjects Enterprise XI 3.X

Summary

Crystal Reports can be deployed in a number of different ways. Each deployment is dedicated to solving specific business problems, and has different capabilities. This document describes the scalability features and intended use of both component and server based reporting engines, so customers can make an informed technology choice.

Author: Blair Wheadon

Company: SAP

Created on: 2 February 2009

Author Bio

Blair Wheadon works as a Product Manager in the Volume Business Unit at SAP. His team works on Crystal Reports, Crystal Reports Server, and the Volume Xcelsius Products (Present, Engage, and Engage Server).

Table of Contents

Summary

Crystal Reports can be deployed in a number of different ways, such as:

- Embedding a report engine into an application (component reporting)
- As a separate reporting server
- As part of a broader Business Intelligence server.

Each deployment is dedicated to solving specific business problems, and has different capabilities. One of the most important capabilities is scalability. Scalability can be defined as the ability to handle growing amounts of work in a graceful manner.

Component reporting is intended for direct embedding into applications, and is the least scalable deployment model we offer. SAP focuses our development resources on improving scalability in server based deployments.

This document describes the requirements, scalability features, and intended use of both component and server based reporting engines, so customers can make an informed technology choice.

Definitions

TERM	MEANING
Component Reporting Engine	Either the Crystal Reports 2008 .Net runtime engine or Crystal Reports for Eclipse 2.0 runtime engine
Component Reporting SDKs	Crystal Reports .NET SDK .Net Report Application Server SDK (In Process) Crystal Reports Java SDK
Scalability	Ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged
Report Job	A 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.
Request	An action on a report job such as refresh, prompt, next page, last page. Actions can be initiated by either end users or an application.
Thread	A processing thread within the multi-threaded reporting engine to respond to an action.

Component Reporting Engine Sizing Guide

Intended Use

The component reporting engines are intended to:

- Share the hardware with another application that adds significant and primary functionality.
- Be lightweight and easy to embed

Jobs, Requests, and Threads

The reporting engine operates on the concepts of a report job, requests, and threads.

- A report job is a report that is open and may or may not be consuming CPU. For example, an end user reading a report represents a report job. An application user interacting with a report programmatically is also a report job. The maximum number of open report jobs at any given time is limited by memory.
- An open report job may generate any number of requests for reporting services. Requests are triggered by an end user or an application.
- Requests are actions like exporting a report, refreshing a report, rendering the next page of the report.
- An open report job typically generates multiple requests over its lifetime.
- A thread is a request being acted on by the report engine. When the request is received by the reporting engine, it accepts the request if it has a free thread. If it doesn't have a free thread then the request pauses until a thread becomes free. When a request completes, the thread is released, and the next waiting request (if any) are accepted.
- Processing a subreport does not create a new thread or report job.

Estimating the Number of Supported Users

You can use the following methodology to estimate the number of supported users. Because the report engine operates on the concept of a reporting thread, we need to convert the number of estimated users into threads.

To start, we divide the users into 2 sets of users, based on how they use the system.

- Active Users – users who are logged into the system frequently throughout the day averaging one request every 4 seconds.
- Light Users – users who will log into the system infrequently and will view a couple of reports and logout having an estimate of one request every 16 seconds.

To derive the number of threads required to satisfy this level of load, divide concurrent users into these 2 categories and then calculate the percentage of each type.

Example

Assume your application has 100 concurrent users, and you assume that 30% of these concurrent users will be using the reporting feature of your application at any given time. This means there's a total of 30 concurrent users (or *report jobs*) of the component reporting engine. Now we need to calculate how many requests these users will be generating. To do so, let's break down the reporting users into active and light users.

Active Users	30%	9 users
Light Users	70%	21 users
Total	**100%**	**30 concurrent reporting users**

Now we can estimate the total simultaneous requests based on each User Type. There are many ways we can do this. The following process is an illustration only.

The following assumptions have been made about rate of simultaneous use based on user group type:

- Every active user generates 1 request every 4 seconds, or ¼ request per second on average.
- Every light user generates 1 request every 16 seconds, or 1/16 request per second on average.
- 2 requests issues in the same second are considered 'simultaneous'.

Formula to calculate simultaneous user requests for peak load:

(Active users X 1/4) + (Light users X 1/16) = Calculated Simultaneous Requests

- or –

(9 X 1/4) + (21 X 1/16) = 3.6 simultaneous requests at peak load when all user types are logged in.

As you can see, a relatively high number of active reporting users (30) can be supported with a small number of threads.

Threads

To ensure the report engine is used as intended, and to deliver optimal throughput with a minimum of configuration, both runtime reporting engines come preset to support a specific number of threads. The thread limit is set to strike a balance between scalability, reliability, and performance.

The .Net component runtime engine supports a maximum of 3 simultaneous threads per process, and the Java component runtime engine supports a maximum of 5 simultaneous threads per process.

Multiple web application server processes can be configured on each server. The methods for doing this differ between Java and .Net, and are beyond the scope of this article.

These thread limits are often mis-interpreted as a user limit, but that is not the case. There is no preset user limit in the component reporting engine.

Thread Blocking

When a request is received by the reporting engine, if the engine has a free thread it will allocate it to service that request. If there is no free thread, then the request will block on a semaphore until a thread free thread becomes available. No errors will be thrown, but the request will wait for a period of time pending a new thread to free up. There is no internal limit on the number of threads that can be blocked. It is not a firt-in, first-out queue, so there is no guarantee that the first thread that blocks will be the first thread that is serviced when a thread becomes available.

The length of time that a request spends blocked depends on how quickly active threads complete. The time for an active thread to complete depends on a number of factors. In general, slower requests are those that:

- Touch all pages in a report, like exporting and printing.
- Have many subreports, especially subreports in the details section of a report.
- Work with a large amount of data or large reports.

Monitoring Thread Blocking

In the Java component engine, you can configure logging so that it will record when it's blocked waiting on a free thread to become available. This can be used to help determine how much blocking is happening in your system, and whether or not you will benefit from a dedicated reporting server.

The following is a complete sample log4j.properties file that will enable logging in the Java runtime engine. When the engine is blocked waiting on a thread, it will log that event. This file needs to be in the \WEB-INF\classes\ folder to be read by the Crystal Reports for Eclipse runtime engine.

Begin log4j file below>>

```
# log4j.properties

# Default log location
crystal.logs.home=${user.home}

log4j.appender.jpeAppender=org.apache.log4j.RollingFileAppender
log4j.appender.jpeAppender.file=${crystal.logs.home}/jpe.log

log4j.appender.jpeAppender.ImmediateFlush=false
# number of log files to keep before deleting the oldest one
log4j.appender.jpeAppender.MaxBackupIndex=30
log4j.appender.jpeAppender.MaxFileSize=500KB

# Log message layout: date-time [thread] priority category - message lineTerminator
log4j.appender.jpeAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.jpeAppender.layout.ConversionPattern=%d{dd MMM yyyy HH:mm:ss} [%t] %-
5p %c - %m%n


# log4j.rootLogger=ERROR

# Covers all of the reporting engine
log4j.logger.com.crystaldecisions.reports=ERROR, jpeAppender
log4j.logger.com.crystaldecisions.threedg=ERROR, jpeAppender
log4j.logger.com.crystaldecisions.common=ERROR, jpeAppender

# ---> More specific loggers can go here
# Log levels used by Log4j in ascending order are:
# debug, info, warn, error, and fatal

# Examples
#log4j.logger.com.crystaldecisions.reports.formulas=DEBUG
<< End log4j.properties file
```

Conclusion

You should use the estimated number of peak requests as a rough guideline to understand if your application requirements fall within the capabilities of the component engine.

The number of users the component reporting engine can realistically service is a function of the following variables:

- Complexity of reports (number of objects, subreport usage, etc..)
- Size of the reporting user base
- How active the users or application is
- Estimated number of peak simultaneous requests
- How much data is retrieved by the report
- Performance expectations of the user base.
- Growth over time – are you planning to target larger customers with your application?

You can use the sizing guidelines presented here as an approximate guide to understand if your application requirements fall within the capabilities of the component engine.

If your application scalability needs exceed the capabilities of the component reporting engine, we offer 2 other reporting servers available with more scalability features. These are described in the next section.

Choosing the Appropriate Solution

Scalability Features Across The Crystal Reports Product Line

	COMPONENT REPORTING ENGINE	CRYSTAL REPORTS SERVER EMBEDDED	SAP BUSINESSOBJECTS XI
How to Integrate with Applications	Report Engine runs in-process with application server	Dedicated reporting server. Reports are processed outside the application process (out-proc). Applications use a remoting API to embed reports.	Dedicated reporting and Business Intelligence server. Dedicated reporting server. Reports are processed outside the application process (out-proc). Applications use a remoting API to embed reports.
Runtime Engine Download Size	~40 MB (the .Net and Java runtimes are about the same size)	300 MB	2 GB
Multithreaded Engine	Yes	Yes	Yes
Queuing of requests	Yes	Yes	Yes
Serializable state on application tier to enable clustering		Yes	Yes
Serializable state on processing tier to enable clustering		Yes	Yes

	COMPONENT REPORTING ENGINE	CRYSTAL REPORTS SERVER EMBEDDED	SAP BUSINESSOBJECTS XI
Designed to use all CPUs on a Server		Yes	Yes
Flexibility in adapting to load		Low	High
Request routing to multiple report engines on the same or different servers		Yes	Yes
Runs as a dedicated service		Yes	Yes
Can be clustered across multiple physical servers		Yes	Yes
Cache Server			Yes
Unbreakable Page Server			Yes
Data sharing among reports with similar data			Yes

Serializable State on the Application Tier

This feature applies to the ability to serialize the state of the ReportClientDocument object between separate web servers. It is supported by our server products.

Serializable State on the Processing Tier

This feature applies to the ability to serialize the state of the ReportClientDocument object between separate reporting servers. It is supported by our server products.

Uses All CPUs on a Server

This feature refers to the ability to use all CPUs on a machine. The component reporting engines are tuned to share hardware with an application, and as such do not spawn threads at a rate that will consume all CPUs on a typical modern server. The number of CPUs used by Crystal Reports Server Embedded is controlled by a keycode.

Request Routing to Multiple Report Engines on the Same or Different Servers

This feature refers to the ability to forward requests to different server instances that may be located on the same or different physical hardware. Crystal Reports Server Embedded 2008 SP1 supports round robin request routing among active servers.

Runs as a Dedicated Service

Both Crystal Reports Server Embedded and BusinessObjects Enterprise XI 3.X services run as dedicated services and deployed across multiple machines for flexible scaling and configuration.

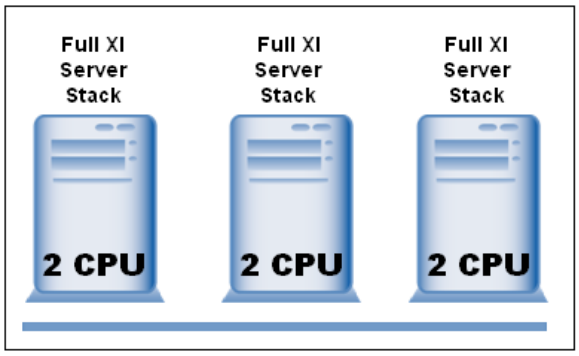
Crystal Reports Server Embedded is simpler in that it runs as a single service for report processing.

BusinessObjects Enterprise XI 3.X has more services to configure independently. They are the:

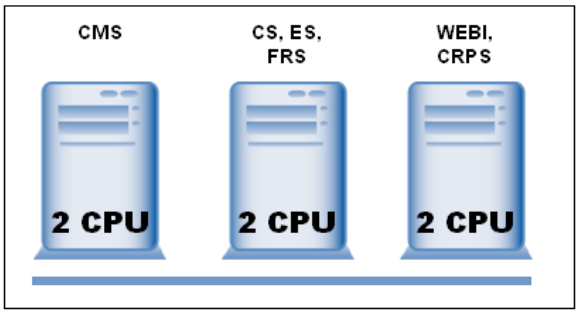
- Cache Server (CS)
- Page Server (CRPS)
- Central Management Server (CMS)
- File Repository Server (FRS)

The following scenarios show how these services can be deployed across multiple machines to achieve the desired combination of redundancy and fault tolerance. This section is intended to communicate the flexibility available in the BusinessObjects Enterprise server, and not serve as a deployment guide. The illustrations refer to other services (like WEBI and ES) which are not used in a Crystal Reports implementation.

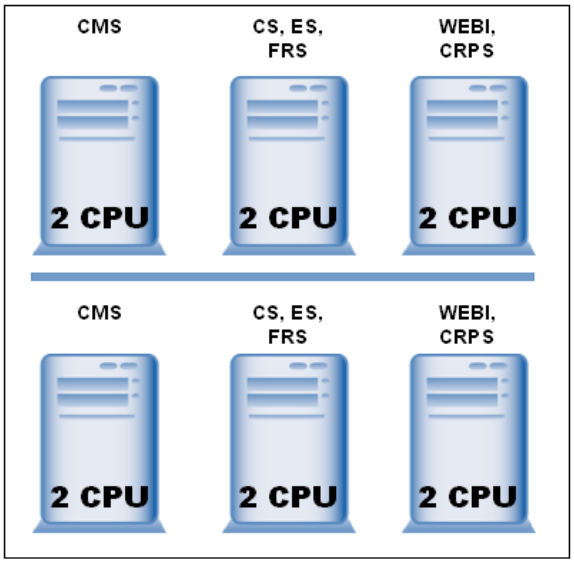
1) **High Redundancy and Fault-Tolerance** - Configure for software and hardware redundancy where the complete BusienssObjects Enterprise XI stack of servers is installed on each of multiple machines (i.e. Datacenter Configuration)



2) **High Performance, Software Redundancy and Fault-Tolerance** - Configure so that the environment divides the BusinessObjects Enterprise processing load in a logical manner, based on the types of work performed by each server. (This is the most common multiple machine configuration)



3) **High Performance, Software/Hardware Redundancy and Fault-Tolerance** - Configure so that the environment divides the BusinessObjects Enterprise processing load in a logical manner, based on the types of work performed by each server, but then duplicates each of these machines to provide for redundancy and fault-tolerance



Page Level Caching

The Cache Server stores report pages generated by the Page Server. By storing report pages in cache, the Page Server and/or database server does not need to be accessed each time the report is requested.

When a report is requested for viewing, the system will first check the Cache Server to see if there are any current available pages cached for that report. If there are pages, the Cache Server will send available report pages to the Web Application Server. If there are no cached pages, the Cache Server will request a Page Server to generate these pages.

The cache server can accommodate approximately 200 simultaneous requests per CPU allocated to the Cache Server service.

Unbreakable Page Server

The Page Server is primarily responsible for responding to page requests by processing crystal reports and generating Encapsulated Page Format (EPF) pages. A single .epf file represents one page of a Crystal report. The Page Server retrieves data for the report from the latest instance or directly from the database (depending on the user's request and user's security level).

Specifically, the Page Server responds to page requests made by the Cache Server. The Page Server and Cache Server interact closely, so cached EPF (encapsulated page files) pages are reused as frequently as possible, and new pages are generated as soon as they are required. The InfoView portal takes advantage of this behavior by ensuring that the majority of report-viewing requests are made to the Cache Server and Page Server. However, if a user's default viewer is the Interactive DHTML viewer, the Report Application Server processes the report.

The Crystal Reports Page Server creates Page Server Sub Processes. Each Sub Process loads the report engine and then instantiates threads or Print Jobs as needed. With the Page Server, if an individual Print Job were to fail for any reason, only those threads contained in the Page Server Sub-Process would be affected. All other Sub Processes within the Page Server service would be unaffected. In addition, individual Sub Processes are shut down after so many requests and a new Sub Process is started, if required, so as to maximize resource management.

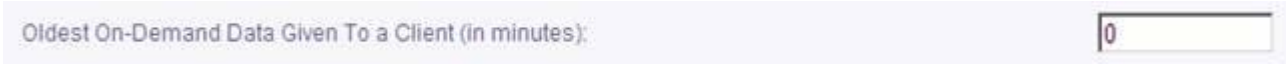
The Page Server can serve approximately 25 simultaneous requests per CPU when run on a dedicated server.

Flexibility in Adapting to Load

The Page Server is designed to run on its own dedicated server, or on a server shared with other services. When run on its own dedicated server, it can serve approximately 25 requests per CPU. When run on a shared server, the number of simultaneous requests can be throttled back to ensure sufficient capacity for other services.

Data Sharing

The "Oldest On-Demand Data Given To a Client (in minutes):" setting in the Central Management Console controls how long the Page Server uses previously processed data to meet requests. If the Page Server receives a request that can be met using data that was generated to meet a previous request and the time elapsed since that data was generated is less than the value set here, then the Page Server will reuse this data to meet the subsequent request.



Reusing data in this way significantly improves system performance when multiple users need the same information. When setting the value of the "oldest processed data given to a client" consider how important it is that your users receive up-to-date data. If it is very important that all users receive fresh data (perhaps because important data changes very frequently) you may need to disallow this kind of data reuse by setting the value to 0. The default is always set to 0 meaning that all users will, by default, receive fresh data.

If Data Sharing can be used in a system, this can decrease the number of CPUs required to process (view) a report.

Scalability Best Practices

Close Report Jobs When Using ReportClientDocument

When writing an application using the component engine, and when you use the `ReportClientDocument` object to modify the report, you should always ensure the `close()` method on the `ReportClientDocument` object gets called to close the report job and release memory.

Avoid page N of M

When designing reports, avoid the ‘Page N of M’ formula as this forces all report pages to be rendered before rendering page 1.

Avoid Subreports in the Details section

When designing reports, avoid putting a subreport in the details section of the RPT template. For example if 1000 records are returned in the detail section of the parent report, and those values are passed to the subreport, then the resulting subreport is invoked and processed 1000 times.

A subreport in a group header can also create similar issues, depending on how many group instances are in your report.

Filter at the data source wherever possible

When designing reports, make the data source do as much filtering as possible before bringing the records into Crystal Reports. The fewer records Crystal Reports has to process, the faster the report will run.

Related Content

Configuring Log4J for Crystal Reports for Eclipse

Copyright