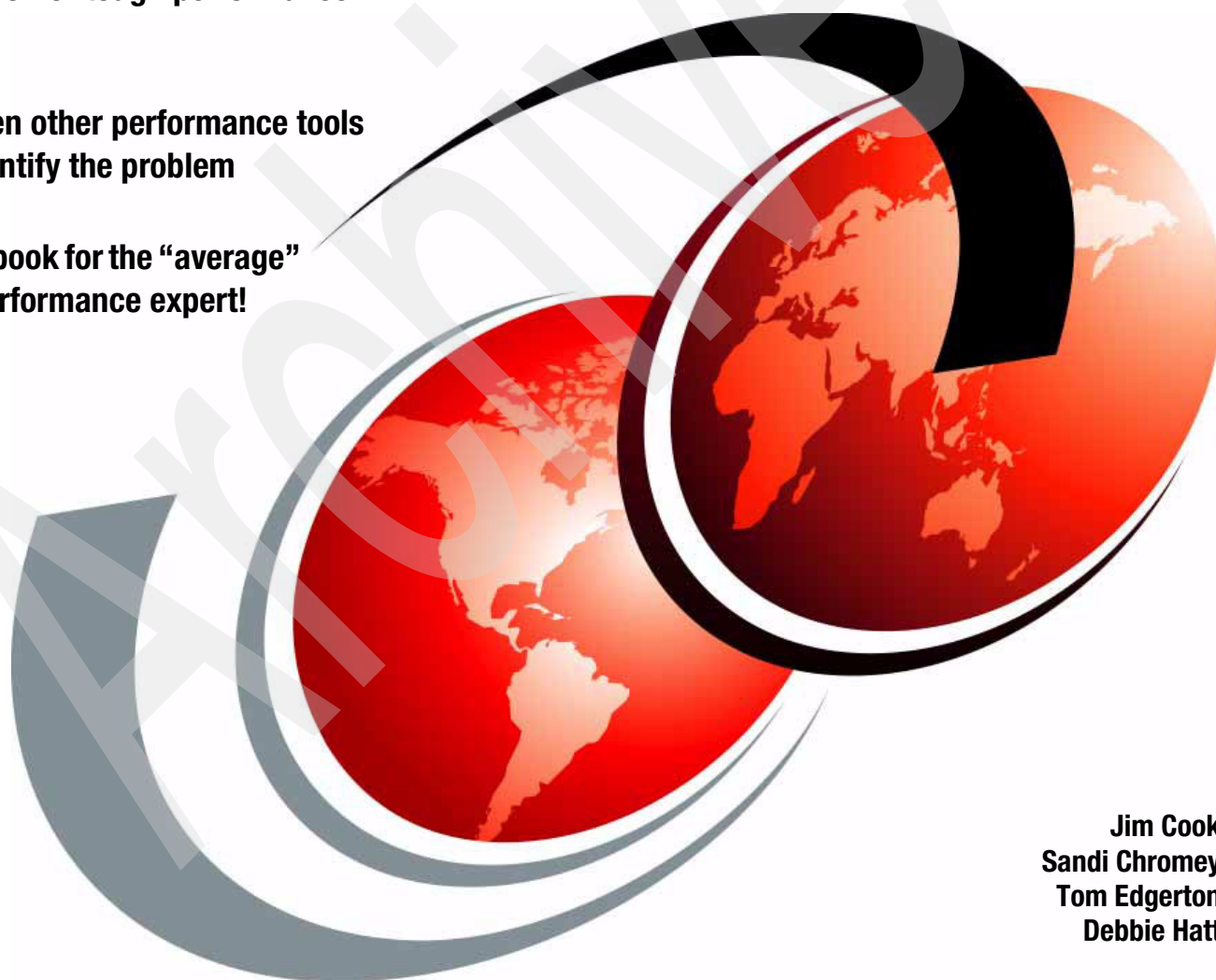


IBM iDoctor for iSeries Job Watcher: Advanced Performance Tool

A tough tool for tough performance problems

Use it when other performance tools do not identify the problem

A how-to book for the “average” iSeries performance expert!



Jim Cook
Sandi Chromey
Tom Edgerton
Debbie Hatt

Redbooks



International Technical Support Organization

**IBM iDoctor for iSeries Job Watcher: Advanced
Performance Tool**

March 2005

Archived

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

Archived

First Edition (March 2005)

This edition applies to Version 5, Release 3, Modification 0 of i5/OS (OS/400), 5722-SS1. It also applies to the IBM iDoctor for iSeries V5R3, Job Watcher Build Level S00052-C00367 and Heap Analyzer Build Level S00024-C00368.

© Copyright International Business Machines Corporation 2005. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Trademarks	viii
Preface	ix
The team that wrote this redbook.	ix
Become a published author	x
Comments welcome.	x
Chapter 1. iDoctor for iSeries Job Watcher positioning	1
1.1 Job Watcher view of IBM-provided iSeries performance tools	2
1.2 Components of performance.	2
1.3 System-level tools	6
1.3.1 Real-time performance tools.	6
1.3.2 WRKSYSSTS command.	7
1.3.3 WRKACTJOB command.	7
1.3.4 WRKDSKSTS command.	7
1.3.5 WRKSYSACT command.	7
1.3.6 Collection Services	8
1.3.7 Management Central	8
1.3.8 Collect performance trace data.	10
1.3.9 Performance Explorer.	10
1.3.10 Database Monitor for iSeries.	11
1.3.11 SQL performance monitors.	11
1.3.12 iDoctor for iSeries suite of products	12
1.4 Performance tools for previously collected data	14
1.4.1 Performance Tools for iSeries Licensed Program product (5722-PT1)	14
1.4.2 Performance Management for iSeries	15
1.4.3 Performance Explorer	16
1.5 Using the system available performance tools	16
1.6 Expanded Job Watcher overview	17
Chapter 2. Overview of job waits and iDoctor for iSeries Job Watcher	21
2.1 Simplified view of running and waiting	22
2.2 Job Watcher terminology	23
2.3 Level set on wait conditions	25
2.4 The mysteries of waiting	26
2.5 Are waits bad?	26
2.6 Detailing waits.	27
2.7 iDoctor for iSeries Job Watcher	29
2.8 Waiting point groupings (wait buckets)	29
2.8.1 Do wait buckets defeat the purpose of many block points?	30
2.8.2 Job Watcher wait points (ENUMs) and wait buckets	30
2.9 LIC queuing primitives and more granular wait points	32
2.9.1 Disclaimer	32
2.9.2 Bucket 1: Dispatched Time (previously referred to as CPU).	33
2.9.3 Bucket 2: CPU queuing.	35
2.9.4 Bucket 3: Total block time.	36
2.9.5 Bucket 4: Reserved.	36
2.9.6 Bucket 5: DASD (page faults)	36

2.9.7	Bucket 6: DASD (non-fault reads)	37
2.9.8	Bucket 7: DASD space usage contention	37
2.9.9	Bucket 8: Idle/waiting for work	37
2.9.10	Bucket 9: DASD writes	38
2.9.11	Bucket 10: DASD (other reads or writes)	38
2.9.12	Bucket 11: DASD operation start contention.	39
2.9.13	Bucket 12: Mutex/Semaphore contention	40
2.9.14	Bucket 13: Journal serialization	40
2.9.15	Bucket 14: Machine level gate serialization	40
2.9.16	Bucket 15: Seize contention	41
2.9.17	Bucket 16: Database record lock contention.	41
2.9.18	Bucket 17: Object lock contention.	42
2.9.19	Bucket 18: Other waits	43
2.9.20	Bucket 19: Main storage pool overcommitment	43
2.9.21	Bucket 20: Java user (including locks)	44
2.9.22	Bucket 21: Java JVM	44
2.9.23	Bucket 22: Java (other)	45
2.9.24	Bucket 23: Socket accepts	45
2.9.25	Bucket 24: Socket transmits	45
2.9.26	Bucket 25: Socket receives.	45
2.9.27	Bucket 26: Socket (other)	46
2.9.28	Bucket 27: IFS pipe.	46
2.9.29	Bucket 28: IFS (other)	46
2.9.30	Bucket 29: Data queue receives	46
2.9.31	Bucket 30: MI queue (other)	46
2.9.32	Bucket 31: MI wait on events	47
2.9.33	Bucket 32: Abnormal contention	47
2.10	Management Central's use of wait bucket mapping	48
Chapter 3. Getting started		51
3.1	Enhancements to Job Watcher at V5R3M0	52
3.1.1	The collection engine	52
3.1.2	GUI	52
3.1.3	Naming	53
3.2	How to obtain Job Watcher	53
3.3	Starting a Job Watcher collection	54
3.3.1	Connecting to your system	54
3.3.2	Access Job Watcher	55
3.3.3	Starting the Job Watcher collection wizard	56
3.3.4	Data collection options	58
3.3.5	Job and task options	63
3.3.6	More on selecting specific jobs and tasks.	64
3.3.7	Ending Options window	74
3.3.8	Job Watcher Wizard summary	76
3.3.9	Active Job Watcher	77
3.4	Stopping a Job Watcher collection	78
3.5	Viewing Job Watcher data	78
3.5.1	Job Watcher view	79
3.5.2	iDoctor preferences.	82
3.5.3	Reports and graphs: Example 1	85
3.5.4	Graphs and reports: Example 2	94
3.5.5	Tabs in the Interval Details window.	97
3.5.6	Additional Job Watcher graphs	108

Chapter 4. Analysis example: traditional batch ILE RPG application	117
4.1 Analyzing the problem	118
4.2 Activation group usage	118
Chapter 5. SQL, call stack, and journal analysis examples	125
5.1 Our example application and preliminary performance analysis	126
5.2 SQL analysis	127
5.2.1 Getting started	127
5.2.2 Viewing an active SQL statement using the GUI	129
5.2.3 Data Viewer	134
5.2.4 Finding jobs running most SQL statements	134
5.3 Call stack analysis for task counts	140
5.3.1 Produce the resolved call stack file	141
5.3.2 The call stack for jobs with active SQL statements	141
5.3.3 Finding user programs running active SQL statements	143
5.3.4 Finding any SQL-related or database-related system activity	143
5.4 Journal analysis	144
5.4.1 No journal caching	145
5.4.2 Journal caching	154
Chapter 6. Analysis example: Java application	159
6.1 Two problem scenarios	160
6.2 CPU “misusage”	160
6.2.1 Job Watcher	160
6.3 Heap growth	172
6.3.1 Heap Analyzer	172
Appendix A. Installing and uninstalling Job Watcher details	191
Installation requirements	192
Installing Job Watcher	192
Uninstalling Job Watcher	197
Server side	197
Client side	197
Appendix B. Database files created by Job Watcher	199
Job Watcher terminology	200
Files created by every Job Watcher collection	202
Files created only when call stack information is specified in a Job Watcher collection	203
Files created only when SQL information is specified in a Job Watcher collection	203
Files created only when activation group information is specified in a Job Watcher collection	204
Files created only when sockets information is specified in a Job Watcher collection	204
Description of ENUM to queueing bucket identifier mappings	204
Description of ENUM to queueing bucket mappings	205
Getting started writing queries over Job Watcher collection data	208
The master file QAPYJWTDE for jobs, threads, and tasks	208
Status file QAPYJWSTS	215
Job file QAPYJWPRC	216
Task dispatchable unit priorities scheme	218
Appendix C. Querying and graphing tips for Job Watcher	221
Querying Job Watcher data	222
Query definition interface	222
Accessing the query definition interface	222

Viewing your query	230
Saving your new query definition	232
Working with query definitions	233
SQL Query view	233
Accessing the SQL Query view	233
Changing field (column) headings in your query results	235
Graph views	238
User-defined graphs	238
Appendix D. Overview of Job Watcher 5250 commands.	241
Job Watcher 5250 commands	242
WCHJOB command	242
WCHJOB parameters	242
CPYJWCOL command	247
CPYJWCOL parameters	247
DLTJWCOL command	247
DLTJWCOL parameters	248
FTPJWCOL command	248
FTPJWCOL parameters	248
RSTJWCOL command	249
RSTJWCOL parameters	249
SAVJWCOL command	249
SAVJWCOL parameters	250
ADDPRDACS command	250
RTVSTKDTA command	250
tRTVSTKDTA parameters	250
Automatically submit a Job Watch	251
Appendix E. Job Watcher advanced topics	255
Collection specification tips	256
Interval size	256
Ensure job names are captured for all jobs on the system	256
Collect data as fast as possible?	257
The Job Watcher collection process in detail	257
Task count state changes	266
Waits that span multiple intervals	267
Job run/wait profile: how to fill in blank intervals	271
Before and after graphs for filled blank interval support	273
Job Watcher limitations	274
Glossary	279
Related publications	281
IBM Redbooks	281
Online resources	281
How to get IBM Redbooks	282
Using the SQL examples in this redbook	282
IBM Support and downloads	282
IBM Global Services	282
Index	283

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law. INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:


This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

DB2®
Domino®
@server®
eServer™
i5/OS™

Integrated Language Environment®
IBM®
iSeries™
OS/400®
POWER5™

Redbooks™
Redbooks (logo) ™
WebSphere®

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM® Redbook is intended for use by those already familiar with most of the iSeries™ IBM-provided performance tools generally available through the i5/OS™ operating systems commands and iSeries Navigator interfaces, and the additional- cost Performance Tools for iSeries, 5722-PT1, licensed program.

iDoctor for iSeries is a set of software performance analysis tools and associated services that go beyond the capabilities of generally available performance-related tools in evaluating the health of your iSeries-based system. These advanced performance-analysis tools are to be used when those generally available performance tools are not able to clearly identify causes for health concerns. These tools gather detailed information and provide automated and graphical analysis of this detailed data.

The Job Watcher is one of these iDoctor tools and is the key “next-step advanced tool” for analyzing this detailed performance data. This book:

- ▶ Gives an overview of most of the IBM-provided iSeries performance measurement and management tools and positions Job Watcher among them.
- ▶ Describes the components of performance and how Job Watcher provides access to detailed performance data.
- ▶ Provides examples of using Job Watcher functions and its graphical interfaces in three environments: a traditional RPG application, an SQL application, and a Java™ application.
- ▶ Provides Job Watcher collected data file/table and field/column definitions, along with examples of SQL queries of this data that provide information beyond that of Job Watcher’s wide array of graphical reports and drill-down information.

The book’s objective is to make the performance analyst proficient in using the Job Watcher as a key tool in their performance analysis tool kit.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Jim Cook is a Senior Software Engineer at the ITSO, Rochester Center. He leads teams that produce iSeries Announcement presentation sets that are maintained on the iSeries technical support Web sites and presents at ITSO iSeries Forums internationally. Jim also produces IBM Redbooks™ about various iSeries and IBM eServer™ i5 topics.

Sandi Chromey is a Senior IT/Architect Specialist with IBM Global Services. She provides iSeries performance support to both internal and external customers within IBM Global Services. Sandi has been with IBM for 23 years, 12 years of which have been in IT. She also has experience in iSeries development and component test.

Tom Edgerton is a Staff Software Engineer with the IBM Rochester Software Support center. He has 20 years of experience in the IT technical support field, the last eight years working at IBM. His areas of expertise include iSeries performance and work management. He writes and teaches extensively about iSeries performance.

Debbie Hatt is from IBM UK Sales and Distribution with extensive experience in analyzing customer performance problems on iSeries-based systems. She holds a first class honours degree in pure mathematics and theoretical statistics from the University of Westminster in London, England. Debbie is proficient in SQL programming and SQL performance analysis. In recent years she has worked in the Rochester development laboratory on iSeries performance tools and with various iDoctor for iSeries advanced performance analysis tools.

Thanks to the following people for their contributions to this project:

Larry Cravens
Steve Fullerton
German Jimenez
Jay Kurtz
Ron McCargar
George Rivest
IBM Job Watcher Development, Rochester

Randy Esch
Aspen Payton
Alexei Pytel
Blair Wyman
IBM Java, Licensed Internal Code, Collection Services Development, Rochester

Rick Turner
iSeries Performance Consultant, retired IBM employee

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will team with IBM technical professionals, Business Partners, and/or customers. Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us because we want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

ibm.com/redbooks

- Send your comments in an e-mail to:

redbook@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

iDoctor for iSeries Job Watcher positioning

In this introductory chapter we discuss the primary components of performance, such as CPU processing time and disk I/O operations, and then summarize the various performance evaluation tools that are available for IBM eServer iSeries and i5 systems, including iDoctor for iSeries Job Watcher.

Specifically, in this chapter we provide:

- ▶ Overview of the components of job or thread performance
- ▶ Brief descriptions of all iSeries performance tools, including the suite of IBM iDoctor for iSeries performance tools, one of which is iDoctor for iSeries Job Watcher
- ▶ Expanded overview of Job Watcher

By reviewing the iSeries performance tools descriptions in this chapter, you can see where iDoctor for iSeries Job Watcher fits among the various performance tools. Some of the tools that are described are included at no cost as part of IBM i5/OS, which is the V5R3 level of OS/400®. Other performance tools described here are available at additional charge.

Terminology notes:

- ▶ In this book, we use the shortened product name, *Job Watcher*, wherever possible.
- ▶ We use the term *iSeries* to represent both iSeries systems and POWER5™-based IBM eServer i5 systems.
- ▶ We generally use *i5/OS* and *OS/400 V5R3* interchangeably.
- ▶ We use the terms *thread* and *task* in many figures and descriptions in this book, but the higher-level term *job name* is used in many of the user interfaces to Job Watcher functions and i5/OS.

1.1 Job Watcher view of IBM-provided iSeries performance tools

You may be reading this book because your system has performance problems, you are curious about iDoctor for iSeries Job Watcher and what it can do for you, or you already use Job Watcher and want more detailed information. Whatever your reason, the need to find out exactly what your system and its applications are doing, learn what resources they are using, and ensure peak performance has never been more critical. To help meet this need, iSeries systems have several readily available *performance tools* to assist you in evaluating your system's performance characteristics. These include i5/OS commands such as Work with Active Jobs (WRKACTJOB) and Work with Disk Status (WRKDSKSTS), i5/OS Management Central's performance monitors, and the command and reports available with the Performance Tools for iSeries Licensed Program Product (5722PT1).

IBM has also developed *advanced performance tools* that are included in the iDoctor for iSeries set of tools. Job Watcher is one of these tools.

Before going into the tools themselves, we review the major components of performance in the next section, "Components of performance." Note that in this topic we sometimes reference the most likely performance tool to use to assess utilization of this component.

Later in this chapter we provide summary-level information about all IBM-provided performance management tools including all iDoctor for iSeries separated advanced performance tools, including Job Watcher.

As the last major topic of this chapter we expand on Job Watcher itself and position this tool versus all of the other performance tools described here. In succeeding chapters we provide more in-depth information about using Job Watcher, based on different types of work examples.

Note that in describing the components of performance, we make reference to the performance tools described in succeeding topics of this chapter.

1.2 Components of performance

This topic describes the performance components of a job, thread, or task running within an iSeries system. We refine the definitions of job, thread, and task later in this chapter.

Job Watcher was designed to give the most accurate accounting of utilized resources and wait states of a job over time on a snapshot interval basis. In other words, you will see the exact elapsed time spent *running* and *waiting*.

The iDoctor graphical view of this data is referred to as the job's *Run/Wait Signature* throughout Job Watcher documentation.

Waits are summarized by type or category and stored in wait buckets for every job, thread, or task running on the system while the Job Watcher is collecting its data. Each wait bucket type has a count and elapsed time, which are continuously, automatically maintained by the system. Job Watcher harvests this wait information at a snapshot interval.

Note that we have used the term *snapshot* several times. This means a point-in-time capture of Job Watcher performance data. A snapshot is not a trace and as such it may miss some small amount of information that a trace would have recorded. However, on a very busy system a trace's collection of detailed information can have a severe negative impact on overall system performance. More detail about Job Watcher's sampling technique is described in Appendix E, "Job Watcher advanced topics" on page 255.

In addition to the wait buckets, the current wait condition of a job, thread, or task (its type and duration, if detected) is also recorded at the end of each snapshot interval.

Additional job information can be harvested at snapshot interval time if requested when starting the Job Watcher. This includes data such as the job's program or procedure call stack (hierarchy of programs or procedures called), the last active SQL statement run, and program or procedure activation statistics. This information further enhances the performance investigation that can be done through the Job Watcher. This kind of data helps identify the most active programs or procedures during the job's, thread's, or task's active run time and what the job was doing.

Several different components may affect performance on an iSeries. A high-level breakdown of these components consists of:

- ▶ CPU
- ▶ Disk I/O
- ▶ Other (seizes, locks, gates)

In the following figures we represent a simplified view of these three main components of performance but do not describe all detailed categories under CPU, CPU & Queuing, Disk I/O, and Other. More details are provided starting with 2.6, "Detailing waits" on page 27.

A further breakdown of CPU components might look like Figure 1-1.

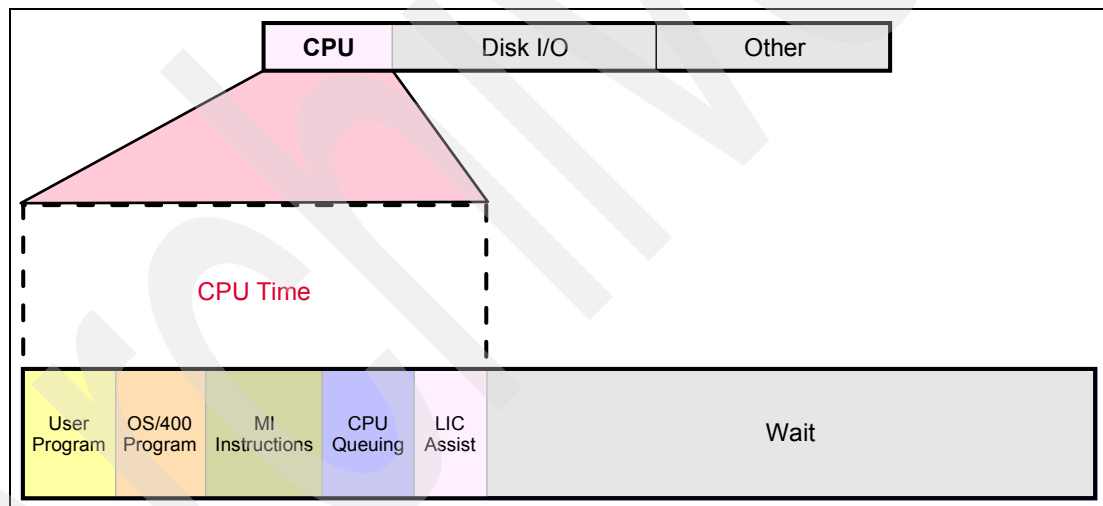


Figure 1-1 CPU components

Note: *LIC assist tasks* are defined as tasks running in the background doing work on behalf of a requesting job. Examples are DBL3xxx tasks that handle asynchronous disk I/Os on behalf of jobs, load/dump tasks (LD component) on behalf of save/restore tasks, and on behalf of other LIC tasks.

We use the term *MI* in these figures and elsewhere to refer to Machine Instruction-level instructions (below the operating system level), which is referred to in more current documentation as the Technology Independent Machine Interface (TIMI).

A further breakdown of Disk I/O components might look like Figure 1-2 on page 4.



A further breakdown of the Other components of performance might look like Figure 1-3 on page 5.

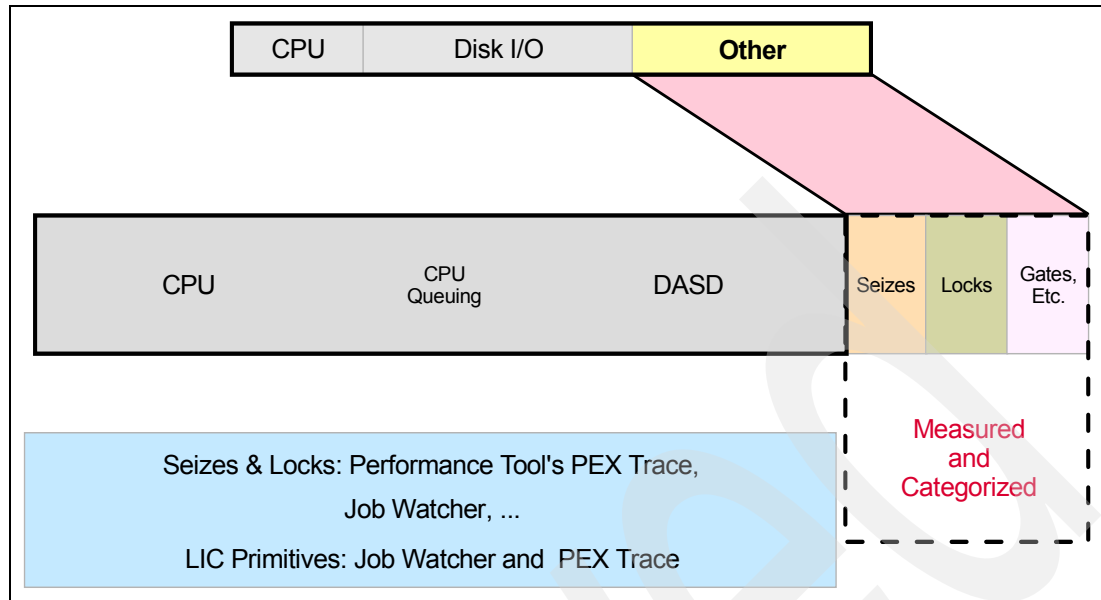


Figure 1-3 Other components of performance

A first look at locks can be determined with the Performance Tools for iSeries Licensed Program Product (5722PT1) Transaction Report options. This processes the performance trace data collected by the i5/OS Start Performance Trace (STRPFRTTC) and End Performance Trace (ENDPFRTTC) commands.

Seizes and gates (a low-level synchronization function) are LIC-level lock/unlock or seize/release mechanisms. iDoctor for iSeries Job Watcher and iDoctor for iSeries PEX analyzer provide more details, including the job/thread waiting on an object, the job/thread holding the object, and the object name and type. Job Watcher measures and categorizes gates (wait and related information). This is all discussed in detail in 2.4, “The mysteries of waiting” on page 26.

Notes: Seizes and gates occur below the TIMI and do not time out. Locks occur above the TIMI and will eventually time out.

Later in this chapter we provide summary information about the iDoctor for iSeries tools mentioned here, Job Watcher, and PEX (trace) analyzer and other performance tools available on iSeries systems. This book focuses on Job Watcher, but also includes the Heap Analyzer in the Java application chapter.

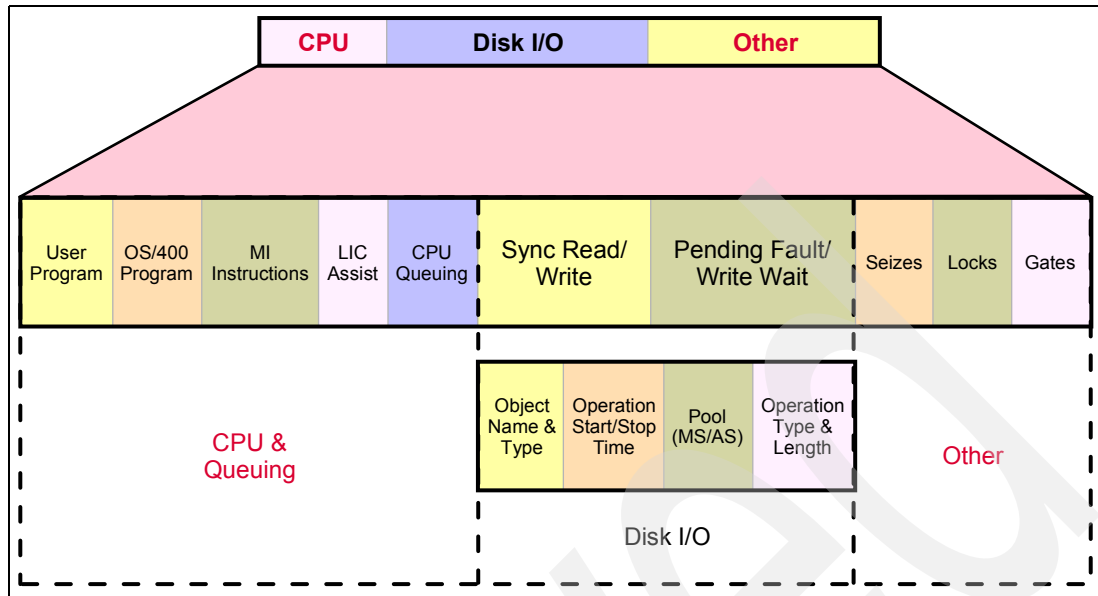


Figure 1-4 Summarization of performance components

Figure 1-4 represents a moderately detailed summarization of the content in the preceding three figures on CPU, Disk I/O, and Other.

One or several of these components can be a bottleneck and a problem source for some transactions. While the thought of replacing the entire iSeries server with a newer and faster model may be an alternative, it might not be the most effective or efficient use of time and resources. With the help of the proper tools, performance can be improved by diagnosing these bottlenecks. By properly balancing system resources, all jobs can run at their optimal level with minimal resource conflicts. Job Watcher and PEX Analyzer measure and present all instrumented components accurately.

There are a variety of iSeries performance tools in addition to those available under iDoctor to help you balance your system resources. Some of these tools show you real-time data and others show you what has already happened on your system. The next section describes the available system-level tools and explains when you would use them.

1.3 System-level tools

Use the following tools to find out the overall use of system resources. Some of these tools show real-time data, and others enable you to analyze previously collected performance data.

1.3.1 Real-time performance tools

Real-time performance tools refer to the tools that can measure system resources such as CPU usage, memory pool page faulting, and disk utilization on a near real-time basis. The tools most often used for system-wide performance monitoring are:

- ▶ Work with System Status (WRKSYSSTS) command (included at no additional charge with OS/400 in earlier releases and i5/OS in its Version 5 Release 3 level)
- ▶ Work with Disk Status (WRKDSKSTS) command (included at no additional charge with OS/400 in earlier releases and i5/OS in its Version 5 Release 3 level)

- ▶ Work with Active Jobs (WRKACTJOB) command (included at no additional charge with OS/400 in earlier releases and i5/OS in its Version 5 Release 3 level)
- ▶ Work with System Activity (WRKSYSACT) command
- ▶ Management Central (included at no additional charge as part of iSeries Access for Windows®, 5722-XE1)
- ▶ Database Monitor for iSeries (included at no additional charge with OS/400 in earlier releases and i5/OS in its Version 5 Release 3 level)
- ▶ SQL Performance Monitors (Visual Explain) (included at no additional charge as part of iSeries Access for Windows, 5722-XE1)
- ▶ iDoctor for iSeries suite of products

Each of these tools is introduced in the following sections.

1.3.2 WRKSYSSTS command

Use the Work with System Status (WRKSYSSTS) command for a quick and easy way to check the status of your system. The Work with System Status display is actually two displays on one screen. The upper half shows statistics about the status of the system, while the lower half is used to both observe and adjust the memory pool sizes and activity levels.

For more information about this command, search for WRKSYSSTS at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm>

1.3.3 WRKACTJOB command

Use the Work with Active Jobs (WRKACTJOB) command to display the performance and status information for the currently active jobs/threads in the system. All information is gathered on a job basis. The jobs are ordered on the basis of the subsystem in which they are running.

For more information about this command, search for WRKACTJOB at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm>

1.3.4 WRKDSKSTS command

Use the Work with Disk Status (WRKDSKSTS) command to observe how the disk units are performing on your system. When viewing the Work with Disk Status display, observe the percent busy data. By looking at the percent busy, you can determine whether there are too few arms or if the arms are too slow.

For more information about this command, search for WRKDSKSTS at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm>

1.3.5 WRKSYSACT command

Use the Work with System Activity (WRKSYSACT) command to view performance data in real-time fashion. This data is reported for any selected job or task that is currently active on the system. The performance statistics reported by this function represent activity that has occurred since a previous collection, and they are reset on every collection interval.

The major differences between WRKSYSACT and the other real-time performance-related display commands are:

- ▶ WRKSYSACT is the only CL command that shows how both the jobs and the tasks use the system resources.
- ▶ WRKSYSACT is not part of OS/400 or i5/OS. It is included as part of the Performance Tools for iSeries Licensed Program Product (5722PT1).
- ▶ Only one user at a time may use the WRKSYSACT command. If this command is already in use in another job, when you issue this command, you get a message indicating this fact.

For more information about this command, search for WRKSYSACT at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm>

1.3.6 Collection Services

Collection Services is a no-charge part of i5/OS that collects a set of iSeries “performance metrics” or “performance categories” over a start/stop time period. The time period start and stop times can be specified by an administrator or automatically when running Performance Management for iSeries (see 1.4.2, “Performance Management for iSeries” on page 15).

Collection Services comes with IBM-supplied, pre-defined sets of performance categories that can be selected for collection or you can select only specific categories to collect under a custom option.

In general, IBM recommends always running Collection Services. When starting Collection Services, you can determine how long to keep a collection object before the system automatically deletes it. This gives you the advantage of having at least summary-level (Graph History) performance data over a long time and collection object data available for detailed analysis (after creating the performance database files based on the collection object).

You can create the performance database files at any time from the collection object as long as it is still on the system.

Starting and ending Collection Services can be performed using either i5/OS commands or the iSeries Navigator interface. More details about Collection Services are provided in 1.3.7, “Management Central” on page 8.

1.3.7 Management Central

iSeries Navigator Management Central provides a far-ranging set of functions that includes performance-related capabilities. Management Central functions are accessed only through the iSeries Navigator interfaces that are part of the iSeries Access for Windows product. The Management Central performance-related functions enable you to:

- ▶ Collect i5/OS performance data using the underlying i5/OS Collection Services function. This data can be used to identify the relative amount of system resources used by different areas of your system. When you collect then convert the collected data to performance database files, you can analyze this information on a regular basis, to help better balance your resources. This analysis can help you detect increased usage in system resources, identify possible sources of performance bottlenecks, or identify jobs or microcode tasks that need further investigation—all with the intent of improving performance from your system. You can customize your data collections to collect only the data you want.

You can also start and stop Collection Services with i5/OS commands, such as Start Performance Collection (STRPFCOL) and End Performance Collection (ENDPFCOL) commands.

When a collection has completed, the compressed collection data is stored within a *collection object*. Some system functions can make use of this condensed set of performance data, but its format is not described externally. You must perform a *create performance database files* function to be able to analyze the collected data.

Information about these files and their fields can be found at the iSeries Information Center at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm>

Expand **Systems management** → **Performance**. To view or download the PDF, click **Print this topic**.

The collection object can be saved, restored, and sent to another iSeries system for later performance database file creation and analysis by the appropriate tools described within this chapter.

- Convert the Collection Services performance data into *performance database files*. This conversion can be an option on the Start Collection Services interface or from a pull-down menu. Alternatively, you can use the Create Performance Data (CRTPFRTA) command. (These files and their fields are defined in the iSeries Information Center.) These files can then be processed by user-written queries or by the Performance Tools for iSeries Licensed Program Product (5722PT1), or other applications that produce performance reports.

A subset of the 5722-PT1 functions can be optional in iSeries Navigator as a *performance graphics* plug-in. The plug-in functions include graphical representations of several performance metrics and producing the performance tools printed reports. When you use the iSeries Navigator plug-in interface you can select performance data from already collected data or the currently running collection, up to the last collected interval that specified when starting the current collection.

It is this performance database file data that you can analyze on a regular basis as described above when collecting the performance data. Using the create performance data interfaces, you can specify subsets of start and stop time periods within the originally specified start and stop collection period. This way you can collect performance data over an extended time period (default is approximately 24 hours) but create performance database files only for the time period you are interested in.

We discuss more about the performance reports in 1.4.1, “Performance Tools for iSeries Licensed Program product (5722-PT1)” on page 14.

- Graphically view summarized collection services data over an extended period of time (one year is the default). This is done using the iSeries Navigator Graph History function. This can be used to observe the variability of specific performance metrics over several months.
- Run *monitors* that can graphically show several performance metrics and other monitor functions in near-real time.

The following types of monitors are provided:

- System monitor
- Job monitor
- Message monitor
- B2B activity monitor
- File monitor

Focusing on performance-related capabilities in this book, both system and job monitors use Collection Services data to track the elements of system or job performance. The system monitors can graphically display in real time the changes to a particular performance metric over time up to approximately the last 60 minutes. Job monitors can monitor some job performance metrics as well as certain job status changes, such as a job starting, ending, or being held.

Depending on the specific monitor type, each type has a unique set of possible automated actions you can specify to take, such as calling a program, holding a job, and responding to a message. You can manually display job details where appropriate.

Through a single iSeries Navigator Management Central workstation, you manage collection services and run the monitors on one or more iSeries systems at the same time. For more details in this area, refer to the iSeries Information Center or see the redbook *Managing OS/400 with Operations Navigator V5R1, Volume 5: Performance Management*, SG24-6565.

1.3.8 Collect performance trace data

You can collect detailed performance trace data using the Start Performance Trace (STRPFTRC) and End Performance Trace (ENDPFTRC) commands. The performance trace data contains more detailed information at the job level than that collected using Collection Services and is placed in separate files.

Collecting this level of detail places additional demands on a running system, so it is often run only for short time periods after you have already collected and analyzed the Collection Services performance database files data. Analyzing this performance trace data can help identify resource contention, program-level resource usage, and transaction delays.

The primary tool for analyzing this trace data is using the Performance Tools for iSeries Transaction Report, described later in this chapter. With this transaction report, you can view the highest CPU utilization programs or the name of a locked object and the associated holder and waiter job names.

See “Performance Tools for iSeries reports” on page 15 for a list of the reports that process this trace data.

If the Performance Tools trace data reports do not provide sufficient details to identify the performance problem, consider using:

- ▶ Performance Explorer
- ▶ iDoctor for iSeries suite of tools

1.3.9 Performance Explorer

Performance Explorer (PEX) is a no-charge i5/OS detailed performance data collection tool invoked by i5/OS commands. This PEX data can be used to help find application program-level causes of performance problems that cannot be identified by Collection Services data and general performance monitors. You define and start or end a PEX collection under one of three options: *STATUS, *PROFILE, and *TRACE.

The Performance Tools for iSeries licensed program provides printed report options for PEX *STATUS and *PROFILE data that includes a low-level summary of disk operations, CPU utilization, file opens, and programs, including MI programs invoked. Printing PEX Profile data can be used to identify actual high-level languages statements (assuming that program compiler statement-optimization levels have not been used on the programs) causing significant CPU utilization.

As described later in this chapter (“PEX Analyzer” on page 13), the iDoctor for iSeries PEX Analyzer provides extended PEX Trace data analysis details, compared to the Performance Tools for iSeries PEX reports. This output can include the PEX information provided by the Performance Tools for iSeries PEX printed data reports plus additional information about wait states and objects referenced details, and DASD space consumption.

1.3.10 Database Monitor for iSeries

Database Monitor for iSeries (DB Monitor) is a tool that can be used to analyze database performance problems that typically is used when you suspect query functions are involved. DB Monitor measures the performance of programs that use SQL to access and manipulate data that may have a significant performance impact.

DB Monitor data is most useful if the user has a basic knowledge of iSeries query optimization techniques.

DB Monitor has two user interfaces:

- ▶ 5250 commands
 - STRDBMON
 - ENDDDBMON
- ▶ iSeries Navigator, which uses the SQL Performance Monitors interface described in 1.3.11, “SQL performance monitors” on page 11.

Attention: When the DB monitor is gathering data, CPU utilization (up to 20 to 30 percent) and disk storage consumption may temporarily increase, especially if there is a significant amount of database activity being collected. On a busy system you would run this monitor for as short a time period as possible.

1.3.11 SQL performance monitors

The iSeries Navigator version of the Start Database Monitor (STRDBMON) function is called a detailed SQL performance monitor. You start this monitor by right-clicking SQL Performance Monitors under the database portion of the iSeries Navigator tree and select **New** → **Detailed**. You can monitor a single query or all queries. After the monitor is started, it appears in the SQL performance monitors list.

SQL Performance monitors provide several predefined reports that you can use to analyze your monitor data. To view these reports, right-click a monitor and select **Analyze**. The monitor does not have to be ended in order to view this information. Additionally, you can import a monitor that has been started by using STRDBMON or some other interface. Right-click SQL Performance monitors and select **Import**. You can then use the same predefined reports to analyze your monitor data.

SQL Visual Explain is a tool used to graphically display the optimization method selected by the database engine (the *optimizer*) when a query is performed. Visual Explain provides a graphical representation and makes it easier for you to determine actions to be taken (such as creating a new index).

Visual Explain can be used:

- ▶ On SQL statements stored in database performance monitor data
- ▶ On detailed monitor data (STRDBMON or Detailed SQL Performance Monitor)
- ▶ Via the Recent SQL Performance Monitors task in iSeries Navigator’s SQL Script Center
- ▶ On SQL statements in the SQL Script Center

Monitor data has to be collected on OS/400 V4R5 or later level of the operating system to guarantee that all of the information is available for representing the data accurately. You may also collect data on one system and analyze it on a different one.

1.3.12 iDoctor for iSeries suite of products

The iDoctor for iSeries software consists of tools to collect performance data on a system and tools to analyze that data either on the same system or on a separate system. The iDoctor suite of tools can be used on a system with performance problems or to determine the health of your system to use as a baseline for future performance problems. In most cases, you would have used the other performance tools described in this chapter before using one of the iDoctor for iSeries components described here.

iDoctor for iSeries currently consists of the following components:

- ▶ iDoctor for iSeries Job Watcher
- ▶ iDoctor for iSeries Heap Analysis Tools for Java (also known as Heap Analyzer)
- ▶ iDoctor for iSeries PEX Analyzer
- ▶ Performance Trace Data Visualizer (PTDV)

The server-side portion of these components consists of a variety of data collection and analysis programs designed to consolidate performance data in a more usable format. The client-side components for Job Watcher, Heap Analyzer, and PEX Analyzer consist of graphical interfaces (also called the graphical user interface, or GUI) for displaying the server data in flexible graph and table views. PTDV provides table and tree views of Performance Explorer Collection data with an emphasis on Java analysis.

PTDV and Heap Analyzer are offered as a free service on an as-is basis. All other components offer a 45-day free trial period with the option to buy after the trial period.

Requests to purchase the components are made via the following Web site:

http://www.ibm.com/eserver/iseries/support/i_dir/idoctor.nsf

Job Watcher

Job Watcher consists of:

- ▶ Tools for collecting data
- ▶ Tools for analyzing and viewing the collected data

Job Watcher returns near-real-time information about a selected set of jobs, threads, or licensed internal code (LIC) tasks. It is similar in sampling function to the system commands WRKACTJOB and WRKSYSACT in which each refresh computes delta information for the ending snapshot interval. In the Job Watcher, these refreshes can be set to occur automatically, even as frequently as every 5 seconds. Better yet, Job Watcher harvests the data from the jobs/threads/tasks being watched in a manner that does not affect other jobs on the system while it is collecting.

Job Watcher collected data includes the following:

- ▶ Standard WRKSYSACT type information
 - CPU
 - DASD I/O breakdown
 - DASD space consumption
 - For jobs or threads, the user profile under which the job/thread is running. For prestarted server jobs that were started under user profile QUSER, you can see the user profile that is currently being serviced by that job/thread, rather than QUSER.
 - And more

- ▶ Expanded details about types of waits and object lock/seize conditions
- ▶ Last run SQL statements syntax
- ▶ Program/procedure call stack, 1000 levels deep
- ▶ More (See 1.6, “Expanded Job Watcher overview” on page 17, for more details.)

Heap Analyzer

Usually the first symptom of a problem related to heap growth (typically found with programs written in C++ or Java) is a gradual or a sudden increased rate of paging in the pool the Java jobs are running in. This is most often caused by an uncontrollable growth of the heap size. If we keep adding memory to the pool, we treat the symptoms but do not remove the problem itself.

Heap Analyzer provides tools for doing the heap analysis, advanced debug, and application review.

Use this tool in two stages:

1. Collect the data.
2. Analyze the data.

Note that you must choose from three collection options:

- ▶ Object table dump
- ▶ Object create profile

PEX Analyzer

The iDoctor for iSeries PEX Analyzer Service includes a software tool specifically tuned for identifying issues affecting system and application performance. The detailed analysis it provides picks up where the PM/400 and Performance Tools products leave off and supplies a drill-down capability that provides a low-level summary of:

- ▶ MI program call flow
- ▶ Program and procedure CPU use
- ▶ Object I/O activity
- ▶ Fault analysis

The client component enables you to condense and graph iSeries PEX trace, statistical, and profile data.

The PEX Analyzer provides graphical visualization of PEX collection data. The Print PEX Report (PRTPEXRPT) command (part of the iSeries Performance Tools Licensed Program Product (5722-PT1)) was the typical method used to view reports created over PEX data. iDoctor for iSeries is a value-add to PRTPEXRPT by providing more flexible interfaces for viewing potentially large and complex data.

PEX Analyzer analyzes PEX TRACE data that was previously collected on the current or some other iSeries 400 system. PEX Analyzer can analyze only data that has been collected. For example, if the PEX TRACE collection did not include database file activity, PEX Analyzer cannot produce reports on database file activity. It is important to understand that it is at the time of PEX data collection that the decision is made concerning which types of data are collected. The decision about what data is collected controls the type of available analyses.

1.4 Performance tools for previously collected data

You must collect performance data (stored within the collection services object) and then, from that collection object, convert the data to the performance database files, if you are planning to use the Performance Tools for iSeries product or write your own queries on those performance database files. There are also third-party performance analysis products that use this performance database files data.

1.4.1 Performance Tools for iSeries Licensed Program product (5722-PT1)

The Performance Tools for iSeries program product includes tools for analyzing the performance database files, and producing information at a system, job, or program level. The Performance Tools for iSeries product provides an interface to the start and end Collection Services functions.

The printed report and graphics functions help you identify and correct performance-related problems. Some commonly used features are outlined in the following sections.

In this section, we assume you understand the previously discussed ways to collect performance data and generate performance database files through either of the following interfaces:

- ▶ From a 5250 (green-screen) display, using commands
- ▶ iSeries Navigator

You may choose to create the performance database files while collecting the performance data. However, to minimize system overhead during the active collection, we suggest that you collect the data and create the files later only when needed.

If you have PM eServer iSeries (also known as PM/400) activated on your system, PM/400 will automatically start Collection Services and you see job QYPSPFRCOL active. If PM/400 is not activated, you can manually start and stop Collection Services through the iSeries Navigator interface, start and end collection options on the Performance tool's GO PERFORM menu or Start Performance Tools (STRPFRT) command, or use the Start Performance Collection (STRPFRCOL) command.

CRTPFRTA command

You can use the iSeries Navigator interfaces to create the performance database files or use the Create Performance Data (CRTPFRTA) command to create a set of database files from the performance information that Collection Services stores in a management collection (*MGTCOL) object. You can also use the Configure Performance Collection (CFGPFRCOL) command and set the Create Database Files (CRTDBF) parameter to *YES. This will automatically create the database files and process the data in the current management collection object as it is collected. The database files are discussed in detail in the iSeries Information Center under **Systems management** → **Performance**:

<http://publib.boulder.ibm.com/pubs/html/as400/v5r1/ic2924/index.htm>

DSPPFRDTA command

The Display Performance Data (DSPPFRDTA) command presents general information about the data collection and provides a summary of the performance statistics. You can use this command to view the performance database files data from a 5250 workstation as an alternative to using the Performance Tools reports.

iSeries Navigator Performance Tools plug-in

Performance Tools for iSeries provides an optional graphical plug-in that can be used by an iSeries Navigator Windows client workstation. This plug-in interface can quickly access any of the sets of performance database files on the system, including up to the last interval of a currently active Collection Services collection. With this plug-in you essentially get a graphical representation of the functions provided by the Display Performance Data command 5250 interface. If desired, you can also print most of the Performance Tools for iSeries reports.

Performance Tools for iSeries reports

Based on the data collected on the system, Performance Tools for iSeries provides you with two sets of reports. From the sample data, you can generate the following reports:

- ▶ System report
- ▶ Component report
- ▶ Job report
- ▶ Pool report
- ▶ Resource report

If you collected the trace data, you may also produce more detailed reports:

- ▶ Transaction report
- ▶ Lock report
- ▶ Batch job trace report

Advisor

Advisor is a semi-automatic tuner that provides an easy-to-use way to improve many of the performance characteristics of your system. Advisor can help you to define specific tuning values and other parts of a processing environment to provide better performance for specific processing conditions on your system. Advisor analyzes performance data and then produces recommendations and conclusions to help improve performance.

1.4.2 Performance Management for iSeries

PM eServer iSeries, a dynamic function integrated into OS/400 and IBM @server i5/OS®, automates many of the functions that are associated with capacity planning and performance analysis. It is simple, requiring nothing beyond activating the function and periodically confirming that data is being collected and transmitted to IBM. It is an automated and self-managing tool that helps keep you aware of any impending performance or capacity system issues. PM/400 uses Collection Services to gather the non-proprietary performance and capacity data from your server and then sends the data to IBM. When you send your data to IBM, you eliminate the need to store all of the trending data yourself. IBM stores the data for you and provides you with a series of reports and graphs that show your server's growth and performance. You can access your reports electronically using a traditional browser.

The PM/400e service includes a set of reports, graphs, and profiles that help you maximize current application and hardware performance (by using performance trend analysis) and to better understand (by using capacity planning) how your business trends relate to the timing of required hardware upgrades such as CPU or disk. Capacity planning information is provided by trending system utilization resources and throughput data which can be thought of as an early warning system for your servers. You may think of PM/400e as a virtual resource that informs you about the health of your system.

PM/400 uses less than 1% of your central processing unit (CPU). It uses approximately 58 MB of disk space, which depends on your hardware model and the size of your collection intervals.

1.4.3 Performance Explorer

Performance Explorer (PEX) is a no-charge i5/OS detailed performance data collection tool invoked by i5/OS commands. This PEX data can be used to help find application program level causes of performance problems that cannot be identified by Collection Services data and general performance monitors. You define and start or end a PEX collection under one of three options: *STATUS, *PROFILE, and *TRACE.

The Performance Tools for iSeries licensed program provides printed report options for PEX *STATUS and *PROFILE data, that includes a low-level summary of disk operations, CPU utilization, file opens, and programs, including invoked MI programs. Printing PEX Profile data can be used to identify actual high-level languages statements (if program compiler statement optimization levels have not been used on the programs) causing significant CPU utilization.

As described earlier in this chapter (“PEX Analyzer” on page 13), the iDoctor for iSeries PEX Analyzer provides extended PEX Trace data analysis details, compared to the Performance Tools for iSeries PEX reports. This output can include the PEX information provided by the Performance Tools for iSeries PEX printed data reports plus additional information about wait states and objects-referenced details, and DASD space consumption.

1.5 Using the system available performance tools

If you suspect your system requires tuning changes, investigate the items outlined in this section. Many of the performance tools described in this chapter provide information about the following sets of information, but Job Watcher provides more complete information (except for some of the other iDoctor for iSeries tools also described in this chapter).

CPU utilization

Use the following tools to determine whether there are too many jobs on the system or if some jobs are using a large percentage of processor time when CPU utilization is running fairly high on your system:

- ▶ Work with Active Jobs (WRKACTJOB)
- ▶ Work with System Activity (WRKSYSACT)
- ▶ Performance Tools for iSeries, System Report

Disk

Use the following tools to determine whether you have enough disk space and disk units on your system:

- ▶ Work with Disk Status (WRKDSKSTS)
- ▶ Performance Tools for iSeries, Component Report

Main storage

Use the Work with System Status (WRKSYSSTS) command to investigate faulting and the wait-to-ineligible transitions.

Communication lines

Use the following tools to find slow lines, errors on the line, or too many users for the line:

- ▶ Performance Tools for iSeries, Advisor
- ▶ Performance Tools for iSeries, Component Report
- ▶ iSeries Diagnostic Tools, Communications Trace
- ▶ Start Communications Trace (STRCMNTRC) command

IOPs and IOAs

Use the following Performance Tools to determine if the workload distribution between the I/O Processors (IOPs) and associated I/O Adapters (IOAs) is not evenly balanced or if there are not enough IOPs and IOAs:

- ▶ Performance Tools for iSeries, Advisor
- ▶ Performance Tools for iSeries, Component Report

Object locks

Use the following tools to investigate locks on your system:

- ▶ Performance Tools for iSeries, Lock Report
- ▶ Work with Object Locks (WRKOBJLCK) command
- ▶ iDoctor for iSeries, Performance Explorer

Program or procedure CPU utilization and disk I/O counts

Use the following tools to investigate locks on your system:

- ▶ Performance Tools for iSeries, Transaction Report (requires STRPFRTTC/ENDPFRTTC collected data)
- ▶ iDoctor for iSeries, Performance Explorer (using Performance Tools for iSeries PEX reports)

You have probably had a chance to use some or even all of these performance tools at one time or another. They have more than likely helped you find lots of problems. In this book we assume you have used the performance tools described above, except the iDoctor for iSeries suite of tools. These other tools have not quite identified the cause of poor performance results, and the next logical step is to use the iDoctor for iSeries Job Watcher.

1.6 Expanded Job Watcher overview

Job Watcher provides the user with the ability to collect many different types of information about a set of jobs (or all jobs and tasks) on a system *and* it does this while providing near real-time, detailed, and summarized views of job, thread, and task performance data that enable you to begin your analysis immediately. The data is collected by a server job, stored in database files, and displayed on a client via the iDoctor GUI presentation facilities.

Job Watcher is a sampling tool similar in function to the i5/OS commands WRKACTJOB and WRKSYSACT in that each refresh computes delta information for the ending snapshot interval. Refreshes occur automatically and can be as frequent as every 100 milliseconds (the *NODELAY—termed *FULLSPEED prior to V5R3—option for one job could even be faster). Job Watcher collects the data in a manner that does not affect the code path of the jobs/threads/tasks being watched. Job Watcher's collection methodology is similar to that of the Work with System Activity (WRKSYSACT) command, which is included as part of the Performance Tools for iSeries Licensed Program Product 5722-PT1.

In 1.2, "Components of performance" on page 2, we describe the components of job performance on almost any operating system, but from an iSeries viewpoint. Job Watcher was designed to give an accurate accounting of these components for a job on a snapshot interval basis. In other words, you will know the exact elapsed time spent running and waiting on a per-job basis. The iDoctor graphical view of this data is referred to as the job's *Run/Wait Signature*.

Waits are summarized by type or categories and stored into wait buckets for every job, thread, and task running on the system. Each wait bucket has two parts, a count and a time, and are

continuously maintained by the system automatically. Job Watcher harvests the wait buckets at the end of each snapshot interval. In addition to the wait buckets, the current wait condition of a job, thread, or task (its wait type, wait duration, and for object lock/seize waits, the object name and its holder if known) is also recorded at the end of each snapshot interval.

There is even more detailed job information besides waits that can be harvested at snapshot time if requested. Data such as the job's call stack, the last active SQL statement executed, and activation group statistics further enhances Job Watcher performance investigation possibilities. This type of data helps identify the programs and procedures that are involved during both the active run time and the wait time of a job, thread, or task.

Whether you are trying to find a problem or determine the potential for improving a job's run time, you should consider using the Job Watcher module of the iDoctor suite of tools and services to complement the other performance tools you are using and the collection routines you already have in place.

In addition to Job Watcher being a monitor and collector of job performance related information, it is primarily used as an analyzer and presenter of the collected data. Its output used for analysis is produced through SQL-based queries and iDoctor GUI presentation facilities (both of which can be modified and saved for later use, as user-defined queries and graphs tailored to specific requirements). Job Watcher also has a powerful function that can launch programs and commands based on user-defined rules and actions. This capability was not available with the level V5R3 Job Watcher that was used while this redbook was created. This function is a very powerful feature and will be made available in a future update to V5R3 Job Watcher.

Job Watcher has been described by some as a Run/Wait profiler because of the characteristic appearance of its graphical data presentation format. By collecting data over important jobs or applications during periods of good performance and viewing and comparing those profiles (or run/wait signatures) with profiles collected at other times when performance changes occurred, you can shorten both the discovery and the remedy cycles. Underneath the profiles are drill-down details that may, in themselves, be sufficient to address the situation appropriately or, at the very least, help to guide you to the next step with additional tools or instrumentation.

Those proficient with Job Watcher usage like it because they can get started into analysis and problem determination after the first two snapshot intervals are complete. Sometimes there is enough detail to recognize victims and nail culprits given the sketch provided by the snapshot intervals without having to run, dump, and analyze traces. Even when the Job Watcher sketch is deemed insufficient and the experts want more detail, Job Watcher and a PEX trace can be run at the same time.

Someone with minimal performance skills can become engaged in the performance collection and initial analysis with some basic training. While they may not be capable of explaining the reasons behind the run/wait signatures, they will likely be able to identify the jobs, quantify the impact to a job's run time or a user's response time, and get the right expert or application developers involved sooner and with more specificity about the situation using Job Watcher data. This initiative can be a key factor in shortening the problem-determination process.

Even though Job Watcher gives a snapshot interval view of job performance and not a detail PEX trace view, the ability to rapidly take snapshots (if needed) and the capability to capture drill-down details at snapshot time gives Job Watcher trace-like tendencies. Often, the Job Watcher approach provides just enough detail to address the situation satisfactorily.

Here are more details about how Job Watcher works and the kinds of information that Job Watcher provides, in general, more completely than other available performance tools.

Job Watcher returns near-real-time information about a selected set of jobs, threads, and licensed internal code (LIC) tasks. It is similar in sampling function to the system commands WRKACTJOB and WRKSYSACT, where each refresh computes delta information for the ending snapshot interval. In Job Watcher, these refreshes can be set to occur automatically, even as frequently as every five seconds. Better yet, Job Watcher harvests the data from the jobs/threads/tasks being watched in a manner that does not affect other jobs on the system while it is collecting.

Job Watcher collected data includes the following:

- ▶ Standard WRKSYSACT type information
 - CPU
 - DASD I/O breakdown
 - DASD space consumption
 - For jobs or threads, the user profile under which the job/thread is running
For prestarted server jobs that were started under user profile QUSER, you can see the user profile that is currently being serviced by that job, rather than QUSER.
 - And more
- ▶ Some data not available anywhere else *in real time* (after the fact, Performance Tools performance trace and Transaction report can identify locked objects and associated job holders and waiters):
 - Seize time (this includes objects being locked at the operating system level and objects being seized at the microcode level)
 - Holder and waiter job and thread
 - Specific Licensed Internal Code block point ID (described in Chapter 2, “Overview of job waits and iDoctor for iSeries Job Watcher” on page 21)
 - Wait occurrences in addition to seizes, including a breakdown of the types of waits (all waits) that occurred
 - Details about the current wait:
 - Duration of the wait
 - Object being waited for
 - Conflicting job/thread/task information
 - Specific LIC block point ID
- ▶ Call stacks (sometimes this is termed the *invocation stack*) that is 1000 levels deep of called programs and procedures
- ▶ SQL statements, host variables
- ▶ Communications data
- ▶ Activation group statistics

The data created by the tool is summarized in many different types of reports and graphs via the GUI client. The client provides a quick picture of what is happening on a per-thread basis when multiple different threads are being watched. You have the flexibility to select a job or an interval and drill down for the details while the watch is in progress or after it has ended.

You may even save the collected data and restore it to another system and analyze it there.

Tip: Run Job Watcher when your system is running normally and you are not having performance problems. This gives you a health check of your system so that when it does have performance problems, you have a baseline to make comparisons to. You might want to run Job Watcher over your system for a period in the morning, afternoon, and possibly during the evening when you might have more batch-type jobs running.

You could also run Job Watcher over specific key jobs on your system when they are running well for baseline-comparison purposes.

Job Watcher can be downloaded from this Web site:

http://www.ibm.com/eserver/iseries/support/i_dir/idoctor.nsf

Select **Build and download information** on the left-hand side pane and choose the appropriate download option.

A typical situation for deciding to use Job Watcher is for a job that is taking a long time to run but is using hardly any CPU resource and disk I/Os are not particularly excessive. Job Watcher is an excellent tool to help you determine job waits, seizures, and other types of contention. Identifying why a job or multiple jobs or threads are not doing anything when they should be, is a primary situation to demonstrate a key set of Job Watcher capabilities. Thus we start with the topic of job waits in Chapter Chapter 2, “Overview of job waits and iDoctor for iSeries Job Watcher” on page 21.

Overview of job waits and iDoctor for iSeries Job Watcher

This chapter gives a general discussion of wait analysis, then focuses on the wait analysis capabilities built into Job Watcher.

Wait analysis is a major feature of the Job Watcher tool. Therefore, in order to understand the examples shown in the other chapters in this book, we explain waits and the Job Watcher wait buckets in detail in this chapter.

Later in this book we include examples of user-written queries against the files produced during a Job Watch collection. To understand those queries and write your own queries, refer to wait bucket definitions listed in Table 2-3 on page 30 of this chapter and the Job Watcher file and field descriptions included in “Job Watcher terminology” on page 200.

2.1 Simplified view of running and waiting

One of the main purposes of the iDoctor for iSeries Job Watcher tool is to, in near real time, quantify the amount of *wall clock time* a thread or task spends running and the amount of time it spends waiting. This first section introduces the concepts of running and waiting in a rather simplistic environment. The real world complexities of iSeries processor sharing, LPAR considerations, and so forth, will come later in our wait bucket descriptions.

All units of work in a system at any instant in time are in one of three states:

1. On a CPU (also known as *dispatched to a processor, running, or active*)
2. Ready to use CPU, but waiting for a processor to become available (also known as *ready or CPU queued*)
3. Waiting for something or someone (also known as *blocked, or idle*)

Terminology note: On an iSeries system, a unit of work can be a single-threaded job, a thread within a multi-threaded job, or a single system (microcode) task. So either *thread* or *task* represents the smallest unit of work that is actually dispatched by the system to run within a processor. In this book we use the terms thread and task interchangeably except where noted.

Even though we use the terms *thread* and *task* in many figures and descriptions in this book, the higher-level term *job name* is used in many of the user interfaces to Job Watcher functions and i5/OS. Just keep in mind that it is a thread or a task that is doing the work and affected by any wait condition.

Hardware multi-threading (iSeries SSTAR technology) and simultaneous multi-threading (eServer i5 POWER5 technology), where a single processor is logically treated as more than one physical processor, raises the issue of differences between the time dispatched to a processor and actual CPU time. Both hardware multi-threading and simultaneous multi-threading are designed to minimize physical processor wait time when threads are dispatched to the processor, but at the microsecond level they have to wait to access instructions or data in the L2 or L3 caches or main memory.

More on this later.

A thread's progress over time of running and waiting is termed its *run/wait signature* and might be characterized as shown in Figure 2-1, with each of the components repeating over time (not shown).

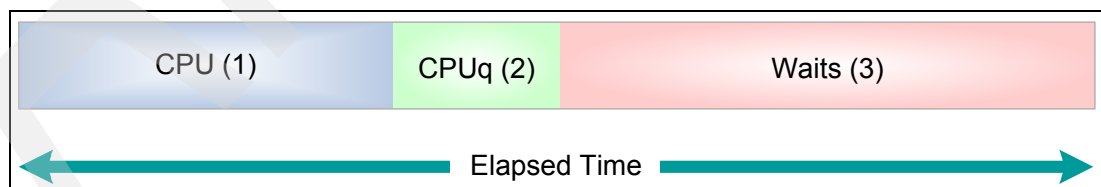


Figure 2-1 A thread's run/wait signature

Attention: The figures in this book that have bar and line graphical displays use multiple colors that may not be completely distinguishable in black-and-white printed format. To interpret these figures properly, view them online in PDF format in real color.

How much time a unit of work spends in state 1 depends on the program design and how much work it is requested to perform.

How much time a unit of work spends in state 2, ready to run but waiting for a processor to use, is a function of the amount of CPU competition it experiences while getting its work done.

How much time a unit of work spends in state 3 depends on many factors. At this point, we need to differentiate between two types of waits:

- ▶ Type A: Waiting for a work request to arrive (also known as *idle*)
- ▶ Type B: Waits that occur while performing a work request (also known as *blocked*)

For example, type A waits in interactive work are considered key/think time. These waits are typically not a problem; any problem is usually external to the machine they are observed on (for example, a communications problem causing slow arrival of work requests).

Note: Traditional batch jobs rarely have any type A waits. A non-interactive job, such as a server job, could have a type A wait. An example is a server job waiting on a data queue or a message queue when the queue is empty.

Type B waits are the interesting ones. While it is debatable whether all of these types of waits should be considered problems, the following is a safe and valid statement:

Outside of CPU usage and contention, type B waits are the reason jobs and threads take as long as they do to complete their work.

A more refined run/wait signature for an interactive job or thread might look like Figure 2-2.



Figure 2-2 Interactive job/thread run/wait signature

A typical batch job type/thread run/wait signature looks similar to Figure 2-3.



Figure 2-3 Batch job run/wait signature

2.2 Job Watcher terminology

To understand Job Watcher graphs and reports, you should be familiar with some terminology, which we summarize in the following topics:

- ▶ Thread

A thread is a separate and unique flow of control within a job. A thread runs a procedure asynchronously to other threads running within the job. This term is used at the iSeries and i5 operating system level, also referred to as *above the Technology Independent Machine Interface* (TIMI). The Licensed Internal Code (LIC) or microcode level actually uses the term *task count*.

A single job can have one or many threads. It always has one single primary thread (also called an initial thread) and may have any number of secondary threads.

The work performed by a job is the sum of work performed by all threads within that job plus the work that is done for that job by other system tasks or jobs.

An LIC task does not have threads.

- Task count

This term is used within the LIC and is discussed here because it may appear in detailed Job Watcher information. At the operating system level it is analogous to a thread number but does not have the same value as the thread number at the microcode level. The microcode task dispatcher manages tasks using the task count value.

A task count is a unique numeric identifier assigned by the microcode to every dispatchable unit of work (job, thread, or task). Every job thread and LIC task has a unique task count value, assigned in ascending sequence, starting when the system or partition is IPLed (also termed *started* or *activated*). A task count value is unique within a partition while that partition is active.

A task count is associated with a TDE, which is a control block representing an individual task or individual thread within a job. Within a job, there is one task count that represents the primary thread, and zero to many unique, different task counts represent each of any secondary threads.

Note that at the microcode level each task count has a run priority that was either established at the operating system level (such as via the job's run priority parameter) or according to microcode implementation requirements.

The task count is used in several of the queries we use on Job Watcher collected data throughout this book. It is used, for example, to get a thread's call stack information.

- TDE

This stands for Task Dispatcher Element, which is an LIC control block that anchors every dispatchable unit (thread or task) to be run in the system. The tasking component of the iSeries system uses the TDE when dispatching work (assigning work to run on a processor). Note that many of the performance metrics are scoped to a thread or task in a TDE and harvested by Job Watcher from the TDE.

For this book, assume that a TDE is associated with a single thread of a job or a single LIC task.

The TDE contains both the task count and the accounting data necessary for running the thread or task in priority among other threads or tasks and for basic performance analysis.

While not showing the TDE values, operating system thread values can be seen on several operating system command screens, such as the **Work with Active Jobs** → **Show job details** → **Work with Threads** option display a thread number.

Task count values are shown only on low-level tools, such as Job Watcher output.

The Performance Tools for iSeries Work with System Activity command shows active operating system thread numbers and LIC tasks.

The thread or job information that can be shown using these WRKACTJOB/WRKSYSACT command interfaces includes information stored in the TDE.

- Snapshot

A Job Watcher collection is point-in-time sample data of the chosen job threads and tasks. Note that we have used the term "snapshot" several times. A point-in-time capturing of a snapshot is not a trace and as such it may miss some small amount of information that a trace would have recorded. However, on a very busy system a trace's collection of detailed information can have a severe negative impact on overall system performance.

Appendix E, "Job Watcher advanced topics" on page 255 describes more about how Job Watcher's sampling technique works.

► Interval

The delta time between snapshots. An interval number is stored in Job Watcher files for chronological sequencing. The interval size is based on the data collection interval that is selected.

Note that as Job Watcher processes each individual thread or task TDE, the actual interval time for each one will be unique. In most cases the intervals will all be close to the user-specific interval duration. As Job Watcher data is sampling data (not a trace), this will not always be the case.

► Eye catcher

An eye catcher is a three-character short description of an individual type of low-level wait within the system. It is used to determine why work is not progressing.

For example, an eye catcher of **Rex** represents a wait for a request for an **exclusive** seize and has an ENUM of 101.

► ENUM

An ENUM is how the system internally identifies an individual detailed type of low-level wait. A single ENUM has only one eye catcher; however, a single eye catcher may be associated with several different ENUMs.

An ENUM is also known as a *block identifier*.

See “Description of ENUM to queueing bucket mappings” on page 205 for a list of all possible waits in ENUM sequence, including eye catchers and the corresponding bucket numbers and description for Version 5 Release 3.

► Bucket number

A bucket number is also known as a *bucket identifier*, *wait bucket*, *block bucket*, or *queueing bucket identifier*.

There are 32 separate bucket identifiers for every job/thread and task on the system. Each bucket identifier has two parts:

- Counts
- Durations

At V5R3, there are 199 different ENUMs, each of which represents an individual low-level type of wait. These 199 ENUMs are mapped into the 32 buckets; hence the counts and durations of several related ENUMs are mapped into the same bucket. This grouping into buckets is done to minimize overhead of extra information being stored in the TDE.

For example, the description of queueing bucket 15 is *seize contention*, and 20 different ENUMs are mapped to this bucket identifier. Two of the eye catchers associated with these ENUMs show **Rex** and **Rsh**, which identify the individual types of seize as *exclusive seize* and *shared seize*, respectively.

2.3 Level set on wait conditions

This discussion applies to individual units of work, which are single-threaded jobs and individual job threads. Many modern application engines involve the use of more than one job or more than one thread (or both) to process each transaction. Examples of multiple job applications on the iSeries systems include the HTTP servers and Domino® servers. Several of these HTTP server and Domino server jobs are also multi-threaded.

The ideas presented in this topic apply in these multi-threaded cases, but each unit of work must be analyzed individually. There is an additional burden placed on the analysis process to

tie together the flow of work across the multiple jobs or threads. With many of today's modern transaction engines it is difficult to differentiate between type A and type B waits.

2.4 The mysteries of waiting

The waiting component of a job or thread's life is easy to compute but rarely discussed and scrutinized.

For batch type work the wait component is computed as follows:

$$\text{Waits} = \text{Elapsed Time} - \text{CPU Time}$$

For interactive type work the wait component is computed as follows:

$$\text{Waits} = \text{Elapsed Time} - \text{CPU Time} - \text{Key/Think Time}$$

Notes:

- ▶ This is a simplified example that excludes CPU queuing for both interactive and batch type work. CPU queuing could be part of a performance bottleneck; it is discussed later in this book.
- ▶ Non-interactive server jobs that wait for new entries on a data or message queue can be characterized by the interactive type work wait components example above. The Key/Think Time would be replaced by "Wait on queue" time.

What is the reason why waits have historically been ignored unless they become so severe that the elapsed time difference becomes painfully obvious? The suggested answer is because little instrumentation and few tools exist to measure and provide details about waits. Waits are the "slightly-off relative" who lives in the basement. Unless his demand for food becomes excessive, or the music gets too loud, he is best ignored. You certainly do not want to talk about him with friends.

2.5 Are waits bad?

This section contends that the answer is yes. (Obviously, we are talking about type B waits.) There is a common misconception that a job or thread that uses high CPU is intrinsically bad. It *might* be bad. For example, if a work process normally 2 hours to complete with 45 minutes of CPU and after a software or data change it now takes 4 hours with 3 hours of CPU, that *is* bad. But just looking at a job or thread (in a non-comparative way) that uses a high percentage of CPU and declaring it bad misses the point that the lack of or minimal occurrences of type B waits is a good thing. For batch-type work (that does not have type A waits, where it is waiting for work to arrive), if the type B waits are reduced or eliminated, the job/thread's CPU density increases. Ultimately, it could use 100% of a processor.

Note: If a single thread consumes all of a single processor for a period of time, it is 100% CPU dense. If it consumes 1/8 of a processor for the same period, it is 12.5% CPU dense. This is true regardless of the number of processors on the system or in the partition. For systems with more than one CPU in the partition, CPU density is not what is seen on WRKACTJOB or WRKSYSACT commands but can be computed by those or from knowing how many CPUs are available to the job.

UDB DB2® multitasking can make a job/thread appear to use more than 100% of a processor as the background assisting tasks promote their CPU consumption numbers into the client job/thread. This can also make accurate capacity planning more difficult.

Here is an example to explain the considerations just discussed:

A batch job runs for 6 hours and uses 117 minutes of CPU. The first thing to consider is how much of the wasted 243 minutes of elapsed time was spent in CPU queuing (that is, contending or waiting for a processor). This chapter demonstrates how this value, and all the waits, can be measured in great detail. But for this example, we suppose that 71 minutes of CPU queuing was involved. This means that the job was in type B waits 172 minutes. This means that the job could potentially run in 3 hours and 8 minutes if the type B waits were completely eliminated. Contrast this with how the job might perform if the CPU speeds on the machine were doubled. One would expect the CPU minutes and CPU queuing minutes to be halved yielding a job run time of 4 hours and 30 minutes. So in summary, eliminating the type B waits could have the job run in 3 hours and 8 minutes. Doubling the CPU capacity could have the job run in 4 hours and 30 minutes.

In summary, wait analysis and reduction can be a very powerful, cost-effective way of improving response time and throughput.

A last word on the significance of waits comes from an IBM On Demand Business poster spotted outside the iSeries Benchmark Center in Rochester, Minnesota, which contains this phrase:

All computers wait at the same speed.

Think about it.

2.6 Detailing waits

Up to this point, this chapter has made the case that wait analysis (and resulting corrective actions) could lead to great satisfaction. What is the first step in wait analysis? It begins with obtaining details about the individual waits.

Remember a summary run/wait signature for a typical batch type job/thread might look like Figure 2-4.



Figure 2-4 Run/wait signature for a batch job/thread

Wait analysis begins by identifying details in the waits (B) component. Look at Figure 2-5.

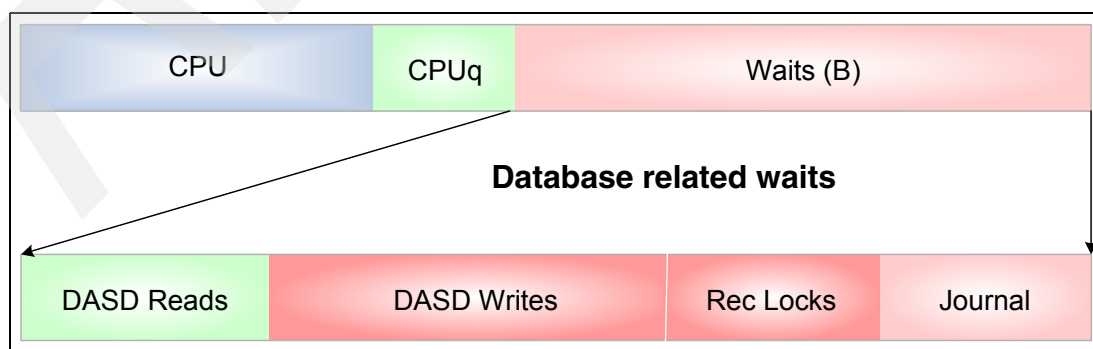


Figure 2-5 Raw amount of time spent in different types of waits

Figure 2-5 on page 27 represents the first phase of detailing: the raw amount of time spent in different types of waits. The next obvious metric needed is the number of each type of waits.

Table 2-1 Number of each type of wait

DASD reads	DASD writes	Record locks	Journal
3,523	17,772	355	5,741

Computed averages are next. Suppose the durations, counts, and averages are like those shown in Table 2-2. Duration and averages are shown in seconds.

Table 2-2 Computed averages

DASD reads	DASD writes	Record locks	Journal
42s	73s	45s	44s
3,523	17,772	355	5,741
0.012s	0.004s	0.126s	0.007s

This is already enough information to begin contemplating actions. Some of the questions to consider:

- ▶ How many of the DASD reads are page faults?
- ▶ Would main memory/pool changes help?
- ▶ What objects are being read from DASD?
- ▶ What programs are causing the reads?
- ▶ How could those DASD reads be reduced, eliminated, or made asynchronous?
- ▶ Could the DASD read response time be better?
- ▶ What objects are being written to DASD?
- ▶ What programs are causing the writes?
- ▶ How could those DASD writes be reduced, eliminated, or made asynchronous?
- ▶ Could the DASD write response time be better?
- ▶ What DB2 files are involved with the record locks?
- ▶ What programs are requesting the record locks?
- ▶ What other jobs/threads are causing the record lock contention?
- ▶ What files are being journaled?
- ▶ What journals are involved?
- ▶ Are the journals needed and optimally configured?
- ▶ Could Commit cycles or the Journal PRPQ be used to reduce this wait component?
- ▶ Is the DASD I/O subsystem write cache (or caches) large enough?
- ▶ Is the DASD configuration well balanced in terms of IOPs, IOAs, busses, RAID, and mirroring configurations?

While these questions must be addressed, it is beyond the scope of this book to delve into details of how to tackle the wait corrective actions.

2.7 iDoctor for iSeries Job Watcher

All preceding material was a generic discussion of wait analysis. Now we focus on the wait analysis capabilities that are built into Job Watcher.

Job Watcher is a sampling-based performance tool. At specified time intervals, or as fast as possible, a Job Watcher command or function will sample anywhere from 1 thread or job to all threads and jobs on an iSeries i5/OS partition (or single system when there are no partition configurations). This gathers a large variety of performance data, much of it beyond the scope of this book. But one of the main reasons for the creation of Job Watcher was to capitalize on wait metrics first introduced into the system in Version 5 Release 1 of the iSeries Licensed Internal Code (LIC) and then expanded in later releases.

Remember the statement that a job/thread is either running on a processor, waiting for a processor to become available, or waiting for someone or something? The LIC has assigned an identifier to all points in LIC code that actually enter the wait state. In Version 5 Release 2, there are about 165 wait points. In Version 5 Release 3, there are about 199 such wait points. Each individual wait point is sometimes referred to as an ENUM, which is shorthand for the C++ programming language's enumerated value and simply means a fixed set of items. When a V5R3 job/thread is in the wait state, it is in one of the 199 possible wait points. The current wait of a job/thread can be referred to by the numerical value of the ENUM (for example, 51) or by a 3 character eye catcher that has been assigned to each ENUM (for example, JBw) or by a text string associated with each (for example, JOBUNDLEWAIT).

Note: Some types of waits are identified with greater granularity than other points. For example, locks and seizures have more individual wait points identified than do other types of waits that tend to share block points.

At the actual run/wait microcode level, only LIC code can truly enter a wait. If an application or i5/OS program enters a wait state, it does so within the LIC code it has caused to run on its behalf.

2.8 Waiting point groupings (wait buckets)

A large number of individual wait points is great from a data-empowerment point of view. However, when it comes to keeping track of them on a wait-point by wait-point basis for every unit of work, it presents challenges to efficient implementation. An ideal design would be for each of the possible wait points (199 in V5R3) to have its own set of data associated with it for each unit of work (job, thread, or task). The minimum amount of accounting data that would be needed includes:

- ▶ Occurrence count
- ▶ Total time accumulator

It was determined that keeping 199 pairs of these numbers associated with every job, thread, or task on a machine was simply too much overhead (mainly in the area of main storage requirements).

On the iSeries and i5 servers, a compromise has been reached that allows for a potentially very large number of individual wait points to be mapped into a modest-sized set of accounting data. This set of accounting numbers is called the *wait buckets*. There are 32 such buckets, but three of them (plus one, which is reserved) have special purposes so there are 28 buckets available to map the 199 wait points. Again, these buckets exist on a per-unit-of-work basis.

2.8.1 Do wait buckets defeat the purpose of many block points?

One might ask, “What is the value in having a large number of unique block points (199 in V5R3) if all of this detail will be lost when they are put into 29 wait buckets?” That is a fair question. The real loss of granularity is felt with sampling-based tools such as Job Watcher. But even with Job Watcher, there is good use of the high wait point counts.

At any given instant in time, the full granularity afforded by all the wait points is available to sampling based tools. For example, at this particular moment in time, thread XYZ is waiting in block point ENUM 114 and it has been waiting there for n microseconds.

Trace-based tools, such as the iDoctor for iSeries PEX Analyzer (which is not covered in this book), can see every wait transition and effectively do the accounting on a per-ENUM basis, making full use of the granularity provided.

For these two reasons, maximizing wait point granularity is a good thing to do.

Important:

1. From an implementation viewpoint, the block points identified in this V5R3 level of documentation are subject to some modification in future releases, subject to changed implementation within the system.
2. The 32 wait buckets that are identified for Job Watcher are a superset of the similar *Counter Set nn wait* values shown when using a Management Central System Monitor - Show Properties job details window. The Counter Set nn wait values may be sufficient for identifying that a wait problem exists, but Job Watcher provides much more detailed wait information.

Learn more in 2.10, “Management Central’s use of wait bucket mapping” on page 48.

2.8.2 Job Watcher wait points (ENUMs) and wait buckets

As mentioned earlier, wait accounting is the core functionality of the Job Watcher tool. The LIC supports remapping of ENUMs to buckets. As shipped, a V5R3 iSeries system does not have all 32 wait buckets utilized; all ENUMs are assigned within the first 16 buckets. Job Watcher performs a remapping when it starts to utilize most of the 32 buckets, thereby maximizing the granularity of wait identification.

Note: There is good reason for assigning ENUMs within the first 16 buckets. Collection Services also harvests wait bucket information for every thread on the system. As mentioned earlier, there is a pair of numbers associated with each bucket (occurrence count, total time). By restricting the default mapping to be within the first 16 buckets, Collection Services reduces the number of DASD bytes that are required to hold the job/thread information. Minimizing this amount-per-thread data is very important. Collection Services is designed so that it can be run 24 x 7 x 365. In that type of environment, every data item gathered on a per-thread basis increases the DASD space requirements for Collection Services.

The bucket definitions Job Watcher uses in V5R3 are shown in Table 2-3.

Table 2-3 Wait bucket definitions for Job Watcher

Bucket	Bucket description
1	Time dispatched to a CPU

2	CPU queuing time
3	Total wait time
4	(Reserved)
5	DASD (page faults)
6	DASD (non-fault reads)
7	DASD space usage contention
8	Idle, waiting for work
9	DASD writes
10	DASD (other reads or writes)
11	DASD operation start contention
12	Mutex-semaphore contention
13	Journal serialization
14	Machine-level gate serialization
15	Seize contention
16	Database record lock contention
17	Object lock contention
18	Other waits
19	Main storage pool overcommitment
20	JAVA user (including locks)
21	JAVA JVM
22	JAVA (other)
23	Socket accepts
24	Socket transmits
25	Socket receives
26	Socket (other)
27	IFS pipe
28	IFS (other)
29	Data queue receives
30	MI queue (other)
31	MI wait on events
32	Abnormal contention

Additional details about the buckets and the ENUMs that are assigned to each are included in the following sections.

Important:

1. The bucket number and its assigned wait type description can change between i5/OS releases. This book is based on V5R3. and Job Watcher mappings for V5R3. These assignments may change in follow-on releases.
2. The system licensed code (LIC) has many more individual block points (as described in section 2.9, “LIC queuing primitives and more granular wait points” on page 32) that are mapped by IBM i5/OS software components using internal interfaces to a smaller number of wait types or buckets. Only one mapping can be in effect at one time across the system or partition.
3. However, each i5/OS component, such as Job Watcher or Collection Services, or Management Central System Monitor → Job Details, maps the wait buckets according each component's presentation of wait information to their end user interface.

For example, Job Watcher maps all 32 buckets. Management Central System Monitor-Job Details shows a subset of the 32 buckets and, by default, does not use the mappings that Job Watcher uses.

The mapping at the time the System Monitor - Show Properties job details information is shown uses either the default mappings or the Job Watcher mappings if a Job Watcher collection has been at least started. The default wait bucket mapping is restored at the next system power on or LPAR partition activation.

See 2.10, “Management Central's use of wait bucket mapping” on page 48 for more details.

2.9 LIC queuing primitives and more granular wait points

Each of the 199 block points in the system is some form of approximately 20 different LIC queuing primitives. Individual block points may be reported (that is, assigned an ENUM) as one of the primitives' ENUMs (which is the default assignment), or (preferably) the specific block-owning LIC component can choose to invent another, more descriptive ENUM for the block point.

For example, consider a synchronous DASD I/O READ wait. Though not 100 percent complete, essentially the wait (block) that occurs in a job/thread while a synchronous DASD read is in progress is implemented with an LIC queuing primitive known as a *single task blocker* (eye catcher QTB, ENUM number 4). That is, when LIC blocks a job/thread due to waiting for a synchronous DASD read to complete, it uses a QTB wait primitive/mechanism. If the component that owns this function (Storage Management) had done no further identifying, that is how such waits would report (QTB, ENUM 4). That would be all right except that there are probably a lot of other block points that also use QTB.

Therefore, it would be impossible to differentiate DASD READ blocks from other blocks. Fortunately, Storage Management, realizing how important it is to quantify DASD operation waits, has invented a different eye catcher and ENUM (SRd, 158) that overrides QTB,4. Before you start to read this section about the wait buckets and their ENUMs, you might read the description of bucket 18 first. Bucket 18 contains many of the default LIC Queuing Primitives ENUMs.

2.9.1 Disclaimer

The following discussion includes the opinion of the writer of the white paper that this chapter is based on. It is far less complete than many people (including the white paper author) would like it to be. However, no single person knows all of the nuances of all 199 wait points in

V5R3. Also, in spite of 199 individual points, many of these remain general and generic to some degree, preventing them from categorically being declared normal or bad. This discussion should be viewed as:

- ▶ Potentially in error or somewhat out-of-date as iSeries software changes become available over time.
- ▶ As a starting point, a guideline to interpreting wait points and buckets, but not as the latest, only, or 100 percent complete information.

2.9.2 Bucket 1: Dispatched Time (previously referred to as CPU)

This accumulates the amount of time a thread or task has been dispatched to a processor. Dispatched to a processor means that the thread or task has been assigned a processor, so it can begin execution of machine instructions. No wait points are assigned or mapped because a dispatched thread or task is not waiting. Job Watcher continues to use the heading of CPU for this bucket. Technically, the previous bucket label of CPU is misleading because Dispatched Time frequently differs from CPU used time. This is especially true when running on a system with simultaneous multi-threading (SMT) implementation or in a shared processor pool partition. See the expanded discussion that follows Figure 2-6, which demonstrates the relationship between Dispatched Time and CPU Time.

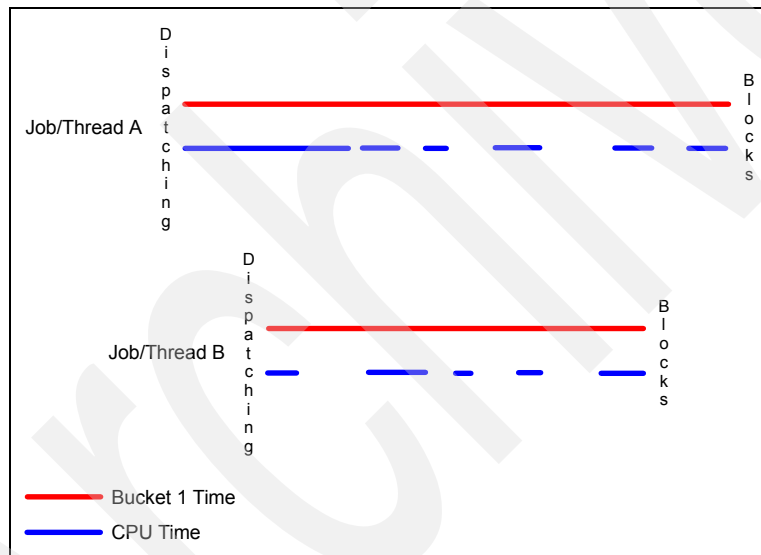


Figure 2-6 Relationship between Dispatched Time and CPU Time

The dispatched time value can differ from the CPU Time measure by other means, including Display Job (DSPJOB), Work with Active Jobs (WRKACTJOB), Work with System Activity (WRKSYSACT part of Performance Tools for iSeries, 5722PT1 product) commands, OS/400 job accounting, and the DELTACPU field in Job Watcher itself). The differences can be significant. The main factors that cause these discrepancies are:

- ▶ The processor hardware multi-threading (HMT) feature of iSeries SSTAR technology systems - 270, 820, 830, 840 models: This can cause bucket 1's time to be larger than the actual CPU time. HMT means that, up to a point in time, more than one thread or task can be simultaneously assigned to the same physical processor. In that scenario, they share the processor's cycles, mainly during long off-chip operations such as memory fetches. Job Watcher's bucket 1 will record the elapsed time a thread or task has been dispatched. The real CPU value will include only the exact number of cycles used by the thread or task while it was dispatched.

- ▶ The processor simultaneous multi-threading (SMT) feature of iSeries POWER5 technology systems - 520, 550, 570, 595 models: This can also cause bucket 1's time to be larger than the actual CPU time. SMT treats a single physical processor as two complete logical processors. Two threads or tasks that are ready to run can be assigned to the same physical processor simultaneously. In that scenario, they share the processor's cycles, mainly during long off-chip operations such as L2, L3 cache, and memory fetches. Job Watcher's bucket 1 will record the elapsed time a thread or task has been dispatched. The real CPU value will include only the exact number of cycles used by the thread or task while it was dispatched.
- ▶ Background assisting tasks, such as those used in the DB2 Multi-tasking feature: Background assisting tasks, which promote (add) their CPU usage back into the client job/thread, will cause the client thread's bucket 1 value to be smaller than the measured CPU time.

Note: This means that bucket 1's value for the assisting tasks themselves would be larger (over time) than the CPU measured by WRKSYSACT or Collection Services. Due to the way these tasks work we do not show their individual CPU utilization, but we do show their associated wait buckets.

- ▶ LPAR shared/partial processors: This is where the tricky concept of virtual processors comes into play. Bucket 1 actually records the elapsed time a thread or task is dispatched to a *virtual processor*, not (necessarily) a physical processor. Similar to HMT and SMT mentioned previously, a virtual processor can be shared across LPARs. If that occurs while a thread or task is dispatched to one of these, the bucket 1 time will be greater than the CPU time, because it will include time the thread or task is dispatched but is waiting for its turn at the physical processor behind the virtual one.

In the simplistic running and waiting introduction to this chapter, Figure 2-7 was presented.

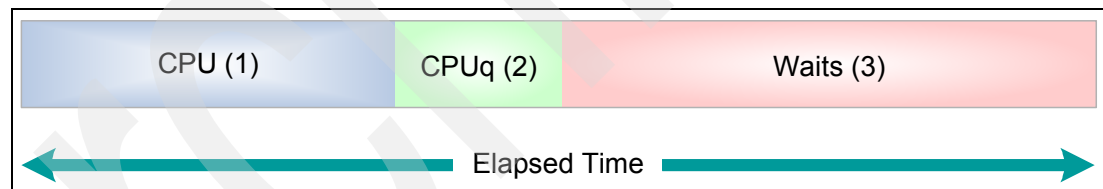


Figure 2-7 A thread's run/wait signature

After the discussion above, the diagram should now look like Figure 2-8.

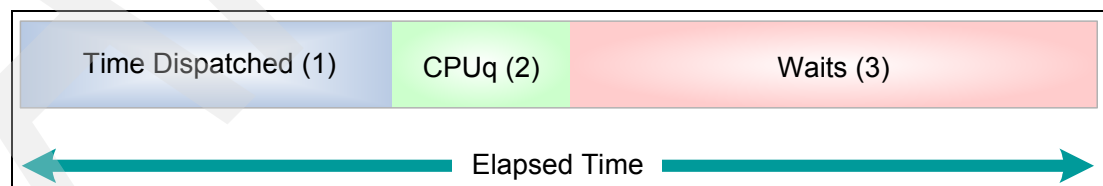


Figure 2-8 A thread's run/wait signature in an LPAR/shared processor environment

A further breakdown of (1), which we can now label Bucket 1, might look like Figure 2-9.

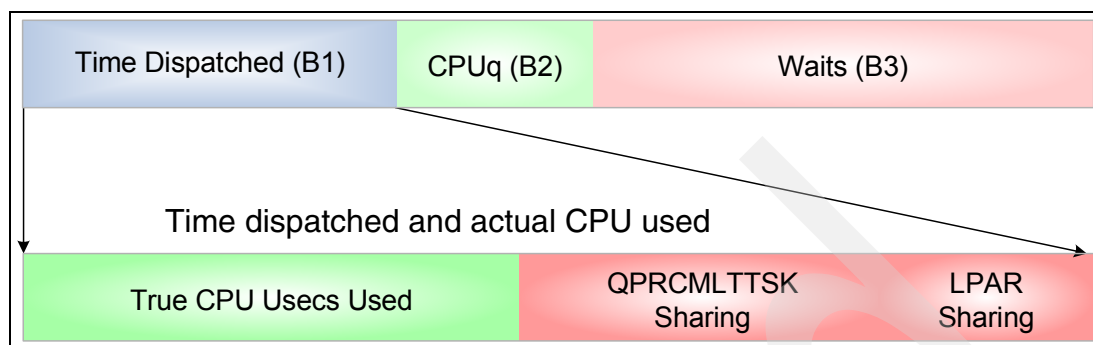


Figure 2-9 Further breakdown of a thread's run/wait signature

Important: The job waits accounting buckets used by Job Watcher do *not* break down Bucket 1 into the constituent parts shown in Figure 2-9. However, much of the time, an additional metric gathered per interval can quantify the first True CPU Microseconds (useconds) Used part. The additional metric gathered is simply the CPU Time Used. (Again, this is a value separate from Bucket 1). When is it *not* true that this additional metric can be used to quantify the True CPU Useconds Used part of Bucket 1? Answer: if the thread being examined is having work done on its behalf by worker/assist LIC tasks.

Those tasks promote their CPU useconds used (and several other metrics, such as DASD I/O counts, exception counts, etc.) back into the requesting thread's metrics. In those cases, the True CPU Useconds Used will be much larger than just what the thread itself had used, making it even more difficult to accurately quantify its CPU usage.

Note: Worker/assist LIC tasks do *not* promote their wait bucket counts and times into the requesting thread. (This is good!)

2.9.3 Bucket 2: CPU queuing

No wait points assigned. Waiting for a processor is a special kind of wait. This is simply the number of microseconds a thread or task has waited, while ready to run, for a processor to become available.

After reading the complex description of Bucket 1 (Time Dispatched), we learned that Time Dispatched can also include “sharing” delays. This raises the natural question:

What is the difference between the QPRCMLTTSK system value on and LPAR processor sharing that is part of Bucket 1 and the queuing that is reported in Bucket 2? After all, queuing and sharing are nearly synonymous.

The simple answer is that there is not much difference. In particular, the LPAR sharing part of Bucket 1 and all of Bucket 2 reflect wall clock time delays in the execution of work due to having to share a processor with other units of work. In the LPAR case, the sharing is occurring with units of work *in other partitions*. Bucket 2 time is the sharing time with other units of work *in the same partition*. QPRCMLTTSK sharing is somewhat more of an accounting shift, rather than true queuing or sharing.

Bucket 1 sharing times versus bucket 2 analogy

Bucket 2 is similar to the time spent waiting in line to gain entrance to a convention hall. Bucket 1's sharing time is the time spent after one is inside the convention, waiting in short lines to view particular booths.

Note: Even if there is little to no actual processor competition (sharing) occurring within a partition while a thread is running, there may always be miniscule amounts of time reported in this bucket. This is an artifact of the fact that some tiny, finite amount of time transpires between when a thread becomes *ready to run* and when it is actually dispatched to a processor.

2.9.4 Bucket 3: Total block time

This is the computed time summary of all the occurrence counts and total block times for buckets 4 through 32.

2.9.5 Bucket 4: Reserved

No wait points assigned.

2.9.6 Bucket 5: DASD (page faults)

These are the waits associated with implicit (page fault) DASD reads.

Page faults are frequently (but not exclusively) caused by having too many jobs/threads running concurrently in too small of a main store pool. If the faulted-on object type is a 1AEF (Temporary Process Control Space), then that is a likely cause. However, there are other types of activity where page faults are expected or normal. For example:

- ▶ When a program or application first starts up in a job/thread.
- ▶ DB2 Access Paths (keyed parts of physical files, or logical files). These tend to be referenced in a highly unpredictable way, and faulting in pages of access paths is considered normal.
- ▶ Page faults on database (DB) and index pages could indicate normal random access to database records.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
161	SFt	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT
162	SFP	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT IO PENDING
164	GRf	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT
165	SRR	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT IO PENDING

2.9.7 Bucket 6: DASD (non-fault reads)

These are simply the waits associated with explicit (read this from DASD for me) synchronous DASD reads. The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
158	SRd	MAINSTORE/LOGICAL-DASD-IO: DASD READ
159	SRQ	MAINSTORE/LOGICAL-DASD-IO: DASD READ IO PENDING

2.9.8 Bucket 7: DASD space usage contention

When an object or internal LIC object is created or extended, and free DASD space has to be located to satisfy the request, there is some level of serialization performed. This is done on an Auxiliary-Storage-Pool-by-Auxiliary-Storage-Pool (ASP-by-ASP) and disk-unit-by-disk-unit basis. Normally one would expect to see few, if any, of these types of waits. If they are present in significant percentages, it usually means that the operating system and licensed microcode are being asked (by applications) to perform a very high rate of object creates, extends, truncates, or deletes.

Note: Opening a DB2 file causes a create of internal control blocks. The size of the DASD space requests is not relevant to these blocks. It is the *rate* of requests that is relevant.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
145	ASM	DASD SPACE MANAGER: CONCURRENCY CONTENTION
146	ASM	DASD SPACE MANAGER: ASP FREE SPACE DIRECTORY
147	ASM	DASD SPACE MANAGER: RR FREE SPACE LOCK
148	ASM	DASD SPACE MANAGER: GP FREE SPACE LOCK
149	ASM	DASD SPACE MANAGER: PERMANENT DIRECTORY LOCK
180	ASM	DASD SPACE MANAGER: TEMPORARY DIRECTORY LOCK
181	ASM	DASD SPACE MANAGER: PERSISTENT STORAGE LOCK
182	ASM	DASD SPACE MANAGER: STATIC DIRECTORY LOCK
183	ASM	VIRTUAL ADDRESS MANAGER: BIG SEGMENT ID LOCK
184	ASM	VIRTUAL ADDRESS MANAGER: LITTLE SEGMENT ID LOCK
185	ASM	DASD SPACE MANAGER: IASP LOCK
186	ASM	DASD SPACE MANAGER: MOVE CHAIN
187	ASM	DASD SPACE MANAGER: HYPERSPACE LOCK
188	ASM	DASD SPACE MANAGER: NON PERSISTENT DATA LOCK
189	ASM	VIRTUAL ADDRESS MANAGER: TEMPORARY SEGMENT ID RANGE MAPPER LOCK
190	ASM	VIRTUAL ADDRESS MANAGER: PERMANENT SEGMENT ID RANGE MAPPER LOCK
191	ASM	VIRTUAL ADDRESS MANAGER: IASP SEGMENT ID RANGE MAPPER LOCK

2.9.9 Bucket 8: Idle/waiting for work

These are the waits on the Technology Independent Machine Interface (TIMI) queue associated with each OS job that is internally known as the MI Response Queue. Normally for 5250-type interactive applications this would reflect the key/think time. Other possible uses would be APPC/APPN SNA-type communication I/O waits.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
340	QMr	IDLE WAIT, MI RESPONSE QUEUE WAIT

2.9.10 Bucket 9: DASD writes

These are the waits associated with synchronous DASD writes, or waiting for asynchronous DASD writes to complete.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
167	SWt	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE
168	SWP	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE IO PENDING
170	SWp	MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WRITE
171	GPg	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE
172	GPP	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE IO PENDING
174	GTA	MAINSTORE/LOGICAL-DASD-IO: GENERIC ASYNC IO TRACKER WAIT
175	GTS	MAINSTORE/LOGICAL-DASD-IO: GENERIC SINGLE TASK BLOCKER WAIT
176	GTT	MAINSTORE/LOGICAL-DASD-IO: GENERIC TIMED TASK BLOCKER

2.9.11 Bucket 10: DASD (other reads or writes)

The ENUMs with the DSM eye catcher deal primarily with actions taken to do DASD unit configuration and setup and should rarely be seen in production jobs or threads.

The other ENUMs with an eye catcher other than DSM are DASD op waits that cannot be differentiated by read or write type of operations. These should rarely occur.

The following list shows the ENUMs that are associated with this bucket.

Enum	Eye Catcher	Description
60	DSM	DASD MANAGEMENT OPS: FIND COMPRESSION GROUP
61	DSM	DASD MANAGEMENT OPS: DEALLOCATE COMPRESS GROUP
62	DSM	DASD MANAGEMENT OPS: READ COMPRESSION DIRECTORY
63	DSM	DASD MANAGEMENT OPS: WRITE COMPRESSION DIRECTORY
64	DSM	DASD MANAGEMENT OPS: INIT COMPRESSION START REORG
65	DSM	DASD MANAGEMENT OPS: MIRROR READ SYNC
66	DSM	DASD MANAGEMENT OPS: MIRROR REASSIGN SYNC
67	DSM	DASD MANAGEMENT OPS: MIRROR WRITE VERIFY SYNC
68	DSM	DASD MANAGEMENT OPS: READ
69	DSM	DASD MANAGEMENT OPS: READ DIAG
70	DSM	DASD MANAGEMENT OPS: VERIFY
71	DSM	DASD MANAGEMENT OPS: VERIFY DIAG
72	DSM	DASD MANAGEMENT OPS: WRITE
73	DSM	DASD MANAGEMENT OPS: WRITE DIAG
74	DSM	DASD MANAGEMENT OPS: WRITE VERIFY
75	DSM	DASD MANAGEMENT OPS: WRITE VERIFY DIAG
76	DSM	DASD MANAGEMENT OPS: REASSIGN
77	DSM	DASD MANAGEMENT OPS: REASSIGN DIAG
78	DSM	DASD MANAGEMENT OPS: ALLOCATE
79	DSM	DASD MANAGEMENT OPS: ALLOCATE DIAG
80	DSM	DASD MANAGEMENT OPS: DEALLOCATE
81	DSM	DASD MANAGEMENT OPS: DEALLOCATE DIAG
82	DSM	DASD MANAGEMENT OPS: ENABLE AUTO ALLOCATE
83	DSM	DASD MANAGEMENT OPS: DISABLE AUTO ALLOCATE
84	DSM	DASD MANAGEMENT OPS: QUERY COMPRESSION METRICS
85	DSM	DASD MANAGEMENT OPS: QUERY COMPRESSION METRICS DIAG
86	DSM	DASD MANAGEMENT OPS: COMPRESSION SCAN READ
87	DSM	DASD MANAGEMENT OPS: COMPRESSION SCAN READ DIAG
88	DSM	DASD MANAGEMENT OPS: COMPRESSION DISCARD TEMP DATA
89	DSM	DASD MANAGEMENT OPS: COMPRESSION DISCARD TEMP DATA DIAG
150	STv	MAINSTORE/LOGICAL-DASD-IO: SAR NOT SET
151	SRv	MAINSTORE/LOGICAL-DASD-IO: REMOVE
152	SRP	MAINSTORE/LOGICAL-DASD-IO: REMOVE IO PENDING
153	SCl	MAINSTORE/LOGICAL-DASD-IO: CLEAR
154	SCP	MAINSTORE/LOGICAL-DASD-IO: CLEAR IO PENDING
156	SUp	MAINSTORE/LOGICAL-DASD-IO: UNPIN
157	SUP	MAINSTORE/LOGICAL-DASD-IO: UNPIN IO PENDING
177	SMP	MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION
178	SMC	MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION CHANGE

2.9.12 Bucket 11: DASD operation start contention

These waits occur when a DASD operation start is delayed due to a very high rate of concurrent DASD operations in progress at the moment it is requested.

The following list shows the ENUMs that are associated with this bucket.

Enum	Eye Catcher	Description
49	QRR	QURESSTACKMSGPOOL, ABNORMAL DASD OP START CONTENTION

2.9.13 Bucket 12: Mutex/Semaphore contention

These are the block points used by C and C++ programming languages (used by i5/OS, some licensed program implementations, or user-written application programs), usually in the POSIX or Portable Application Solutions Environment (PASE) environments, to implement Mutex and Semaphore waits.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
15	QMG	QUMUTEXGATE, NOT OTHERWISE IDENTIFIED
16	QSm	QUSEMAPHORE, NOT OTHERWISE IDENTIFIED

2.9.14 Bucket 13: Journal serialization

This book holds the waits associated with database or other object (such as data queues) journaling. The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
50	JBo	JOURNAL BUNDLE OWNER WAIT FOR DASD COMPLETION
51	JBw	JOURNAL BUNDLE WAIT FOR DASD COMPLETION
260	EFJ	EPFS: WAIT FOR OS TO FINISH APPLY JOURNAL CHANGES
261	ERJ	EPFS: WAIT FOR OS REQUEST TO APPLY JOURNAL CHANGES

Enum 50 is the wait in the thread that is actually performing the DASD write to the journal. It is the wait for DASD journal writes to complete. Journal uses some fancy approaches to DASD ops to do their writes absolutely as efficiently as possible. That is why DASD writes to journals are not accounted for in the DASD write bucket. (It is good for performance analysis to have these journal writes differentiated).

Enum 51 is the wait that occurs in threads other than the one that's performing the DASD writes. For efficiency, multiple jobs or threads can ride along the journal DASD writes performed by other jobs or threads.

2.9.15 Bucket 14: Machine level gate serialization

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
2	QGa	QUGATE, NOT OTHERWISE IDENTIFIED
3	QTG	QUTRYGATE, NOT OTHERWISE IDENTIFIED

QGa is a very high-performance, low-overhead serialization primitive used by LIC. It is the type of primitive in which there can be one and only one holder. Normally, QGa is used in areas in which the anticipated wait time, if any, is very small (microseconds).

Note: Some related block points (QGb, QGc, QGd) are covered in 2.9.33, "Bucket 32: Abnormal contention" on page 47.

2.9.16 Bucket 15: Seize contention

Think of seizes as the LIC's equivalent of locks. A seize almost always occurs on or against an MI object (DB2 physical file member, data queue, program, library, user profile, and so forth). Seizes can conflict with locks and can cause lock conflicts. There is a large variety of seizes: shared, exclusive, fair, and intent-exclusive. Explaining everything about seizes is beyond the scope of this book; they are, after all, internal LIC primitives that are subject to change at any time. If seizes are a significant percentage of a run/wait signature, examining the call stack, wait object, and holding task or thread (if any) are probably necessary to understand what is causing the contention.

Seizes are frequently (but not exclusively) associated with database objects and operations. Concurrent activities in multiple jobs such as opens, closes, journal synchronization points, access path building, and so forth, might lead to seize waits. Other actions or objects that can experience seize waits include libraries and user profiles during high rates of concurrent create or delete activity in multiple jobs.

This bucket is the first mention of the term *holding task* (or holding thread). However, Job Watcher has the ability to determine the holder for more than just seize waits. It can do so for locks, database record locks, and other wait ENUMs based on a low-level serialization primitive called a *gate*.

In the area of waiters and holders, it needs to be pointed out that the waiter (the job or thread that is experiencing the wait) is frequently the victim, not the culprit. Therefore in a number of situations, it is more important to concentrate on the holders because they are affecting the waiters.

The following list shows the ENUMs that are associated with this bucket.

Enum	Eye Catcher	Description
100	Rex	SEIZE: EXCLUSIVE
101	Rex	SEIZE: LONG RUNNING EXCLUSIVE
102	Rsh	SEIZE: SHARED
103	Rix	SEIZE: INTENT EXCLUSIVE
104	Ris	SEIZE: INTENT SHARED
105	Rfa	SEIZE: FLUSH ALL
106	Rdx	SEIZE: DATABASE EXCLUSIVE
107	Rii	SEIZE: INTERNAL INTENT EXCLUSIVE
108	Rot	SEIZE: OTHER
109	Rlk	SEIZE: LOCK CONFLICT
112	RXX	SEIZE/LOCK IMPOSSIBLE
125	Rsp	SEIZE: OFF-LINE IASP
126	Rra	SEIZE: RELEASE ALL
127	Rrs	SEIZE: RELEASE
133	Rss	SEIZE/LOCK: INTERNAL SERVICE TOOLS HASH CLASS GATE
135	Rmf	SEIZE: MONITORED FREE
141	Rcu	SEIZE: CLEANUP
143	Rsv	SEIZE/LOCK: SERVICE
320	S0o	COMMON MI OBJECT CHECKER: SEIZE OBJECT
321	S0i	COMMON MI OBJECT CHECKER: SEIZE FOR IPL NUMBER CHECK

2.9.17 Bucket 16: Database record lock contention

The typical record lock contention occurs when a holder has the record (SQL row) locked for an update or exclusively for a short period of time.

The following list shows the ENUMs that are associated with this bucket.

Enum	Eye Catcher	Description
110	RDr	DB RECORD LOCK: READ
111	RDu	DB RECORD LOCK: UPDATE
123	RDw	DB RECORD LOCK: WEAK
134	Rxf	DB RECORD LOCK: TRANSFER
136	Rck	DB RECORD LOCK: CHECK
139	Rcx	DB RECORD LOCK: CONFLICT EXIT

2.9.18 Bucket 17: Object lock contention

These are the conflicts between threads involving objects. The OS frequently needs and obtains locks during such operations as:

- ▶ Opening a DB2 file
- ▶ Creating or deleting an object into a library
- ▶ Moving an object to a different library
- ▶ Ownership changes

The operating system can also use *symbolic locks* as a serialization mechanism. These are called *space location locks*.

Application code can explicitly use locks via the Allocate Object (ALCOBJ) command.

The following list shows the ENUMs that are associated with this bucket.

Enum	Eye Catcher	Description
113	RIr	LOCK: SHARED READ
114	RIo	LOCK: SHARED READ ONLY
115	RIu	LOCK: SHARED UPDATE
116	RIa	LOCK: EXCLUSIVE ALLOW READ
117	RIe	LOCK: EXCLUSIVE NO READ
118	RMr	LOCK: SEIZE CONFLICT, EXCLUSIVE
119	RMo	LOCK: SEIZE CONFLICT, SHARED
120	RMu	LOCK: SEIZE CONFLICT, INTENT EXCLUSIVE
121	RMa	LOCK: SEIZE CONFLICT, INTENT SHARED
122	RMe	LOCK: SEIZE CONFLICT, INTERNAL INTENT EXCLUSIVE
124	RMm	LOCK: MATERIALIZE
128	Rdo	LOCK: DESTROY OBJECT
129	Rdp	LOCK: DESTROY PROCESS
130	Rdt	LOCK: DESTROY THREAD
131	Rdx	LOCK: DESTROY TRXM
132	Rar	LOCK: ASYNC RETRY
137	Rtr	LOCK: TRACE
138	Rul	LOCK: UNLOCK
140	Rlc	LOCK: LOCK COUNT
142	Rpi	LOCK: PROCESS INTERRUPT

Note: The ENUMs with the word SEIZE in the description are lock conflicts caused by existing seizures on an object.

2.9.19 Bucket 18: Other waits

This is the Job Watcher's wait accounting bucket for *miscellaneous wait types*.

The following list shows the ENUMs that are associated with this bucket.

Enum	Eye Catcher	Description
1	QCo	QUCOUNTER, NOT OTHERWISE IDENTIFIED
4	QTB	QUSINGLETASKBLOCKER, NOT OTHERWISE IDENTIFIED
5	QUW	QUUNBLOCKWHENDONE, NOT OTHERWISE IDENTIFIED
6	QQu	QUQUEUE, NOT OTHERWISE IDENTIFIED
7	QTQ	QUTREEQUEUE, NOT OTHERWISE IDENTIFIED
9	QPo	QUPPOOL, NOT OTHERWISE IDENTIFIED
10	QMP	QUMESSAGEPOOL, NOT OTHERWISE IDENTIFIED
11	QMP	QUSIMPLEMSGPOOL, NOT OTHERWISE IDENTIFIED
12	QSP	QUSTACKLESSMSGPOOL, NOT OTHERWISE IDENTIFIED
13	QSC	QUSTATECOUNTER, NOT OTHERWISE IDENTIFIED
17	QSB	QUSYSTEMBLOCKER, NOT OTHERWISE IDENTIFIED
240	RCA	LIC CHAIN FUNCTIONS: SMART CHAIN ACCESS
241	RCI	LIC CHAIN FUNCTIONS: SMART CHAIN ITERATOR
242	RCM	LIC CHAIN FUNCTIONS: CHAIN MUTATOR
243	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 1
244	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 2
245	RCE	LIC CHAIN FUNCTIONS: CHAIN ACCESS EXTENDED

In the list, the ENUMs with eye catchers beginning with a Q are the generic wait points. They are the low-level LIC blocks that have not (yet) been uniquely identified. These ENUMs will be seen when LIC blocks code that has not gone out of its way to uniquely identify the block point. The only identification that exists is the differentiation afforded by the type of LIC blocking primitive used. A few opinions can be offered for some of them:

- QCo is frequently used for timed waits. The wait used at the core of the DLYJOB command is a QCo wait. It is also used by POSIX condition variable waits.
- QTB is a wait primitive used for many purposes (unfortunately). About the only generic statement that can be made on it is that it is used when a thread or task is waiting for a specific action to happen on its behalf, explicitly for *that* thread or task.

For example, waiting for synchronous DASD reads and writes to complete uses QTB blocks. Fortunately, DASD reads and writes have further been identified, so they are covered by their own unique buckets. They are not lumped into QTB (see other buckets).

2.9.20 Bucket 19: Main storage pool overcommitment

These waits indicate that one or more main storage pools are currently overcommitted. Regular operations, such as explicit DASD reads or page faults, are being delayed in order to locate free main storage page frames to hold the new incoming data.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
155	GCP	MAINSTORE/LOGICAL-DASD-IO: CLEAR PAGE OUT WAIT
160	GRQ	MAINSTORE/LOGICAL-DASD-IO: DASD READ PAGE OUT WAIT
163	GFP	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT PAGE OUT WAIT
166	GRR	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT PAGE OUT
169	GWP	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE PAGE OUT WAIT
173	SPw	MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WAIT, POOL OVERCOMMITMENT

2.9.21 Bucket 20: Java user (including locks)

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
200	JUW	JAVA: USER WAIT
201	JSL	JAVA: USER SLEEP
203	JSU	JAVA: SUSPEND WAIT
209	JOL	JAVA: OBJECT LOCK
304	JSG	JAVA: SYNCHRONOUS GARBAGE COLLECTOR WAIT
305	JSF	JAVA: SYNCHRONOUS FINALIZATION WAIT

2.9.22 Bucket 21: Java JVM

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
302	JWH	JAVA: GARBAGE COLLECTOR WAIT HANDSHAKE WAIT
303	JPH	JAVA: PRIMARY GC THREAD WAIT FOR HELPER THREADS DURING SWEEP
306	JGW	JAVA: GARBAGE COLLECTOR WAITING FOR WORK
307	JFW	JAVA: FINALIZATION WAITING FOR WORK

Enums in the Java user bucket represent wait points that a customer application or other third-party software can request of a Java Virtual Machine (JVM). The wait points are entered only when the customer application specifically requests the JVM to enter the desired wait. If the waits in this category are very frequent or endure for extended periods of time, the user should investigate the wait and determine if the frequency or duration is something that is unexpected.

Special consideration should be given to the Java Object Lock bucket. Waits in the Java Object Lock bucket should be neither frequent nor long. Frequent Java Object Lock waits are an early indication of an application that will not scale as the number of users is increased. Long Java Object Lock waits indicate an application contention point and reduce the throughput that the application will achieve.

2.9.23 Bucket 22: Java (other)

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
202	JWC	JAVA: WAIT FOR COUNT
204	JEA	JAVA: END ALL THREADS
205	JDE	JAVA: DESTROY WAIT
206	JSD	JAVA: SHUTDOWN
207	JCL	JAVA: CLASS LOAD WAIT
208	JSL	JAVA: SIMPLE LOCK
300	JGG	JAVA: GARBAGE COLLECTOR GATE GUARD WAIT
301	JAB	JAVA: GARBAGE COLLECTOR ABORT WAIT
309	JGD	JAVA: GARBAGE COLLECTION DISABLE WAIT
310	JGE	JAVA: GARBAGE COLLECTION ENABLE WAIT

2.9.24 Bucket 23: Socket accepts

These are socket op block points associated with the socket accept() API call. Normally, but not always, these represent a thread waiting for work.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
210	STA	COMM/SOCKETS: SHORT WAIT FOR ACCEPT
211	LTA	COMM/SOCKETS: LONG WAIT FOR ACCEPT

2.9.25 Bucket 24: Socket transmits

These are waits associated with Socket API calls that are sending and transmitting data.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
212	STS	COMM/SOCKETS: SHORT WAIT FOR TCP SEND
213	LTS	COMM/SOCKETS: LONG WAIT FOR TCP SEND
216	SUS	COMM/SOCKETS: SHORT WAIT FOR UDP SEND
217	LUS	COMM/SOCKETS: LONG WAIT FOR UDP SEND

2.9.26 Bucket 25: Socket receives

These are waits associated with socket API calls that are receiving data.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
214	STR	COMM/SOCKETS: SHORT WAIT FOR TCP RECEIVE
215	LTR	COMM/SOCKETS: LONG WAIT FOR TCP RECEIVE
218	SUR	COMM/SOCKETS: SHORT WAIT FOR UDP RECEIVE
219	LUR	COMM/SOCKETS: LONG WAIT FOR UDP RECEIVE

2.9.27 Bucket 26: Socket (other)

The primary wait points that should be seen from this bucket involve the SELECT socket API. That API can be used by an application for a variety of complex waiting scenarios.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
220	SAS	COMM/SOCKETS: SHORT WAIT FOR IO COMPLETION
221	LAS	COMM/SOCKETS: LONG WAIT FOR IO COMPLETION
222	SSW	COMM/SOCKETS: SELECT SHORT WAIT
223	SLW	COMM/SOCKETS: SELECT LONG WAIT

2.9.28 Bucket 27: IFS pipe

These waits are due to Integrated File System (IFS) pipe operations.

The ENUMs associated with this bucket are:

Eye		
Enum	Catcher	Description
252	PPC	IFS/PIPE: MAIN PIPE COUNT
253	PRP	IFS/PIPE: READ END OF PIPE
254	PWP	IFS/PIPE: WRITE END OF PIPE
255	PRW	IFS/PIPE: PIPE READ WAITERS
256	PWW	IFS/PIPE: PIPE WRITE WAITERS

2.9.29 Bucket 28: IFS (other)

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
250	PRL	IFS/PIPE: FILE TABLE ENTRY EXCLUSIVE LOCK
251	PRC	IFS/PIPE: LIC REFERENCE COUNT

2.9.30 Bucket 29: Data queue receives

These are the waits on MI data queue objects.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
341	QMd	DATA QUEUE WAIT

2.9.31 Bucket 30: MI queue (other)

These are waits on MI queue objects other than the two preceding types. In general, these would be internal operating system operations or user queue/dequeue waits.

Note: For example, most of a subsystem monitor job's normal wait point is a dequeue on an MI queue (that is, neither the MI response queue nor a data queue).

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
342	QMo	OTHER MI QUEUE WAIT

2.9.32 Bucket 31: MI wait on events

Machine Interface event waits are used mainly across jobs by internal operating system programs.

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
330	EMw	MI EVENT WAIT

2.9.33 Bucket 32: Abnormal contention

These waits reflect a high rate of concurrent waits and releases occurring against a wide variety of many of the other wait points listed previously. There are two types of these waits:

- ▶ Unsuccessful wakeup retries (QGb, QGc, QGd)
- ▶ Waiting in line to buy a ticket that gets you into the main wait line (QWL)

The following list shows the ENUMs that are associated with this bucket.

Eye		
Enum	Catcher	Description
8	QRP	QURESSTACKMSGPOOL, NOT OTHERWISE IDENTIFIED
14	QWL	QUWAITLIST, WAITING FOR ACCESS TO A WAIT LIST
40	QGb	QUGATEB, ABNORMAL QUGATE CONTENTION, FIRST RETRY
41	QGc	QUGATEC, ABNORMAL QUGATE CONTENTION, SECOND RETRY
42	QGd	QUGATED, ABNORMAL QUGATE CONTENTION, THIRD RETRY

Now that you are familiar with the concept of waits and wait groupings, you are better prepared to take advantage of the Job Watcher data by using the run/wait graphing support. In the next chapter we explain how to set up, run, and generally navigate through the Job Watcher functions and interfaces.

However, we provide one more topic on waits, from a Management Central System Monitor viewpoint. This is to illustrate the interdependence between System Monitor - Job Details wait information and Job Watcher wait bucket mappings of the same detailed wait "blocks" collected by the system.

The next topic expands on the Management Central mapping of wait buckets as described earlier in the shaded "Important" boxes under:

- ▶ 2.8.1, "Do wait buckets defeat the purpose of many block points?" on page 30
- ▶ Table 2-3 on page 30

2.10 Management Central's use of wait bucket mapping

The job detail information available through the System Monitor-Show Properties function uses a default mapping of the wait bucket information collected by the system's microcode unless a Job Watcher collection has been started at least once.

Refer to the Job Watcher wait buckets listed in Table 2-3 on page 30 while reading this topic.

Collection Services is actually collecting the performance data for the system monitor and uses the mapping in effect each time it collects performance data. Assuming Job Watcher has not been active on your system or partition since the last IPL, Management Central uses the default wait mappings (as Counter Set *nn* wait labels) as shown in Figure 2-10.

Note that the counter set wait label starts at zero, rather than one. Counter set 0 corresponds to Job Watcher waits bucket number 1.

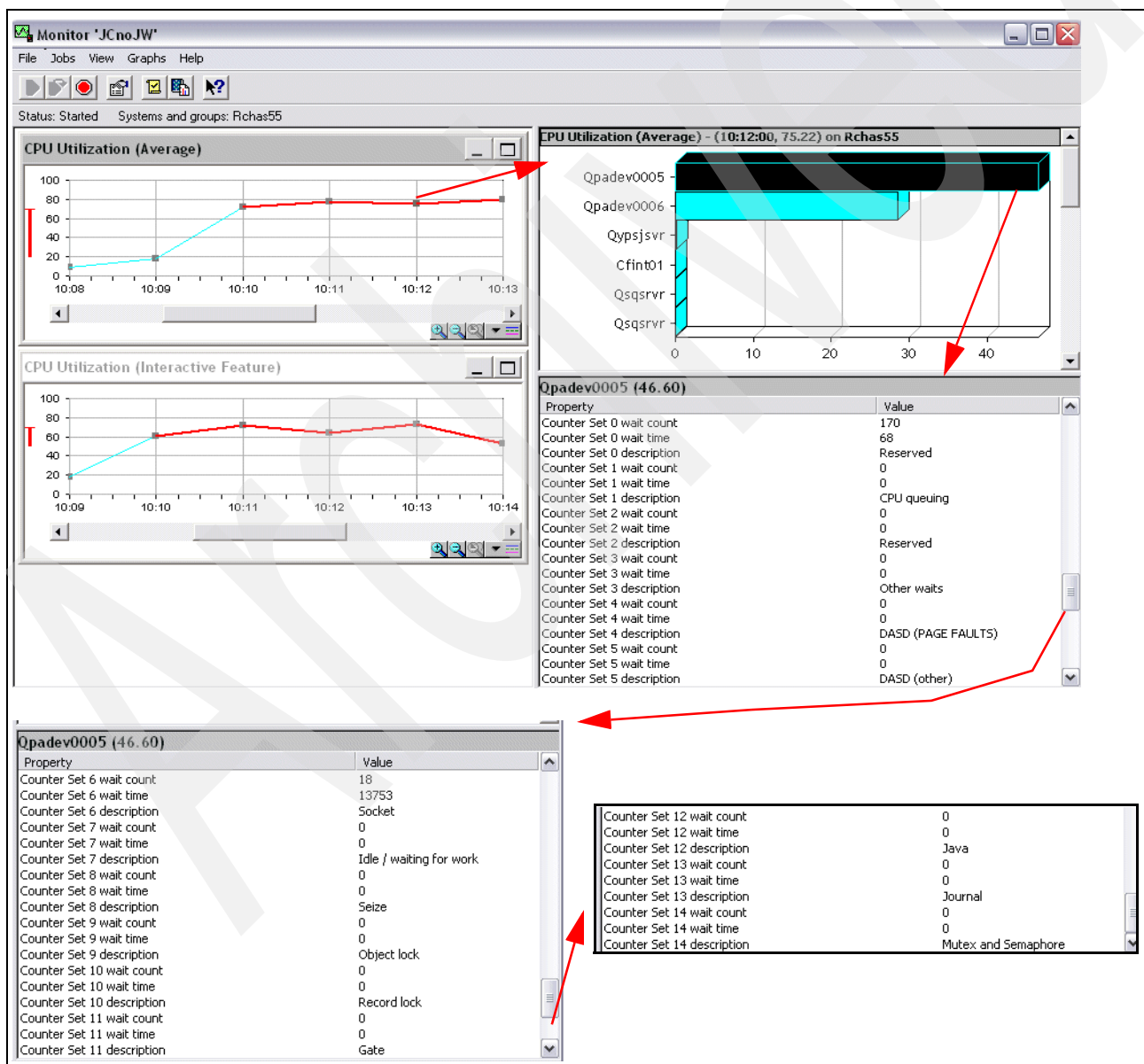


Figure 2-10 System Monitor Job Details default wait buckets mapping

After a Job Watcher collection has been started within your system or partition, Collection Services and the System Monitor use the Job Watcher wait bucket mappings in Show Properties (job details), as shown in Figure 2-11.

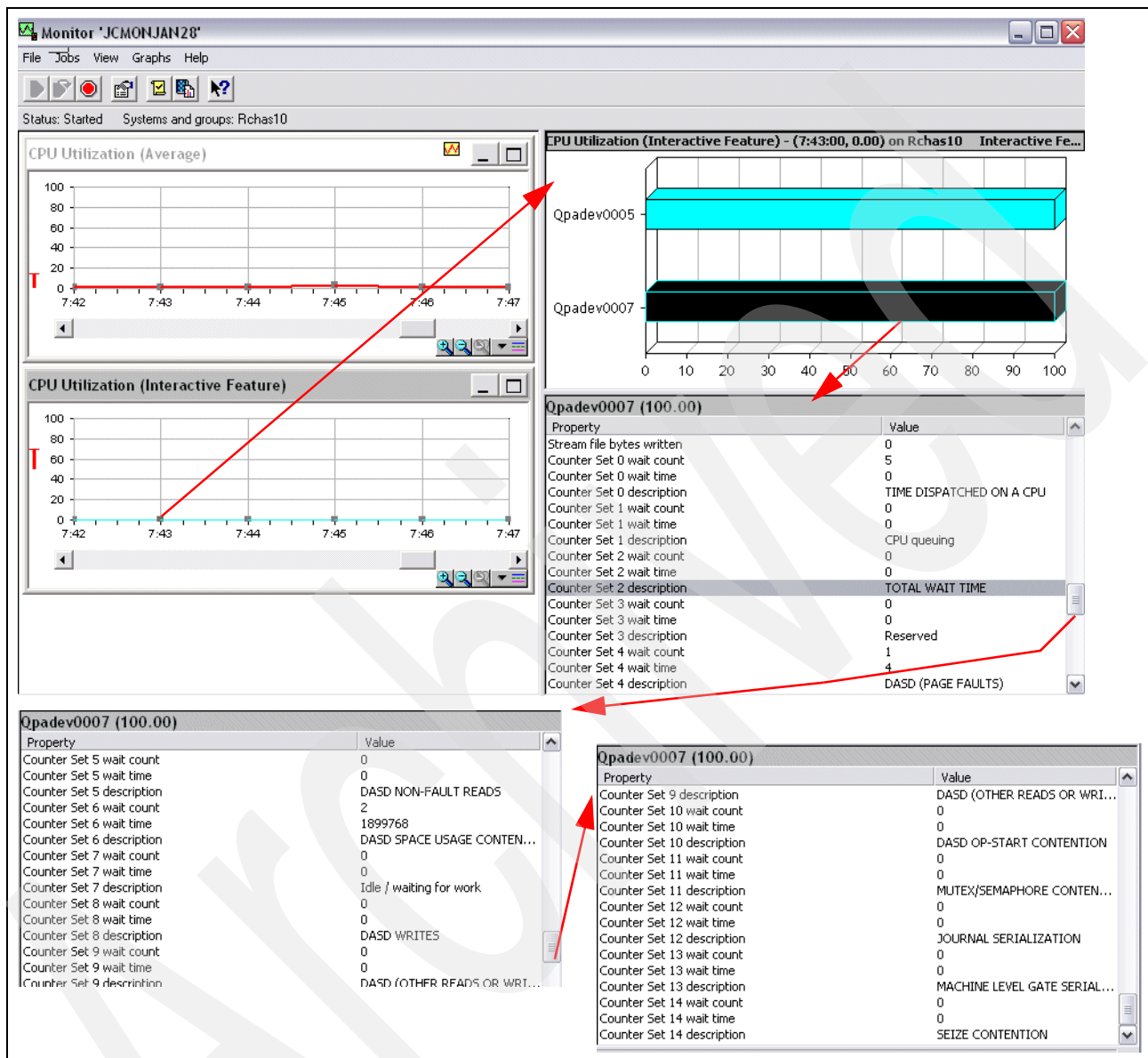


Figure 2-11 System Monitor Job Details after Job Watcher wait buckets mapping

The mapping at the time the System Monitor - Show Properties job details are shown uses either the default mappings or the Job Watcher mappings if a Job Watcher collection has been at least started. If a Job Watcher collection starts while a System Monitor is running, the next refresh of the job detail properties uses the Job Watcher mapping.

The default wait bucket mapping is restored at the next system power on or LPAR partition activation.

Archived

Getting started

In this chapter we cover what is new in V5R3 Job Watcher and familiarize you with how to obtain Job Watcher and navigate through its array of functions. We show various options in the Job Watcher wizard and provide a basic understanding of the more commonly used graphs.

The topics we include cover include:

- ▶ Enhancements to Job Watcher at V5R3M0
- ▶ An overview on how to obtain Job Watcher
- ▶ Starting a Job Watcher collection
- ▶ How to use data collection options
- ▶ Examples of viewing system-wide or job-specific signatures
- ▶ Example of zeroing in on a selected portion of a signature
- ▶ Example of the most commonly used data collections and graphical views of data

Attention: The figures shown in this book that have bar and line graphical displays consist of multiple colors in real life. These figures are best viewed online in real color, rather than in printed format which typically shows only black, white, and shades of gray, to understand what is being described.

3.1 Enhancements to Job Watcher at V5R3M0

We highlight the V5R3 level of Job Watcher usability and functionality improvements and then proceed with to how to obtain Job Watcher, leading to its general capabilities and navigation. If you are not familiar with Job Watcher in previous releases, just read this book!

3.1.1 The collection engine

- ▶ Completely rewritten engine, providing faster and more efficient collection with more options. Reference 3.3, “Starting a Job Watcher collection” on page 54 for more details.
- ▶ Dramatically less data collected per time interval, with corresponding dramatic reduction in the resources consumed by the collecting job. V5R3 Job Watcher adopts a design similar to Collection Services: only a tiny amount of information is put into DB2 files for jobs, threads, and tasks that have been completely idle for a Job Watcher snapshot interval.
- ▶ Data collection only for active threads and tasks, reducing the number of bytes consumed for call stack information.

In the V5R2 version, it was not even possible to collect call stacks for every thread or task, only for those that were running or in certain types of predetermined waits. It would have been impractical to allow all threads or tasks to have their call stacks harvested, because the engine did not differentiate between active and idle threads or tasks.

- ▶ New ways to select which jobs and tasks to collect: system-wide, unique, or generic job or task name, pool ID, subsystem, taskcount, current user profile.
- ▶ Options for dynamically selecting which call stacks to collect; or all can be collected.
- ▶ An option to periodically write Job Watcher data in huge blocks rather than every interval, providing significant collection time performance improvement at the cost of delays in seeing the data.¹
- ▶ Multiple ways to end a collection: DB size limit, number of seconds, number of intervals, selected jobs or tasks become inactive.
- ▶ ASP threshold option added to end collection if total disk space exceeds a specified limit.²

3.1.2 GUI

- ▶ New graphs, for example: transactions, CPU by priority, system-wide seize contention per interval
- ▶ New graph categories to make it easier to locate the data of interest
- ▶ New property pages at summarized (over multiple jobs) and detailed levels (a single job) to consolidate information and make analysis easier
- ▶ New options to manipulate Job Watcher data providing copy, FTP, save, and restore support using the GUI or 5250 commands
- ▶ Easier-to-use Collection Wizard
- ▶ Automatic font resizing in graph views
- ▶ Enhancements to the query definition interface
- ▶ Improved call stack viewer, with information about why the call stack was collected and an option to view the call stack for the holding job if appropriate
- ▶ Database sizes now listed for each Job Watch collection

¹ This is available only when starting Job Watcher using the WCHJOB command from the QIDRWCH library.

² The default is to use ASP threshold and can be changed only when using the QIDRWCH/WCHJOB command.

3.1.3 Naming

All iDoctor component libraries have now been renamed to begin with QIDR:

- ▶ New library QIDRWCH replaces QPYRTJW.
- ▶ New library QIDRGUI replaces QYPBASE.
- ▶ Database files named QAPYJW* instead of QPYRT* ship with OS in library QSYS.

V5R3M0 enhancements descriptions used in this book come from the following Web site:

https://www-912.ibm.com/i_dir/idoctor.nsf/faqNewAtV5R3.html

3.2 How to obtain Job Watcher

If you do not already have Job Watcher installed on your iSeries or i5 server, you can obtain a 45-day trial version by filling out and submitting a trial agreement form located at:

https://www-912.ibm.com/i_dir/idoctor.nsf/JWTrialAgreement.html

For the trial agreement form, you must provide:

- ▶ The serial number of your iSeries or i5 system
- ▶ Your OS/400 version and release level

After the form is submitted, you should receive a reply e-mail within 24 hours that will contain a temporary access code. The access code can be added either during the installation of Job Watcher or after Job Watcher has been installed. The access code must be entered before you can start a Job Watcher collection or analyze an existing Job Watcher collection that has been restored from another iSeries or i5 system.

As Figure 3-1 shows, all iDoctor components including Job Watcher can be downloaded from:

https://www-912.ibm.com/i_dir/idoctor.nsf/downloadsV5R3.html

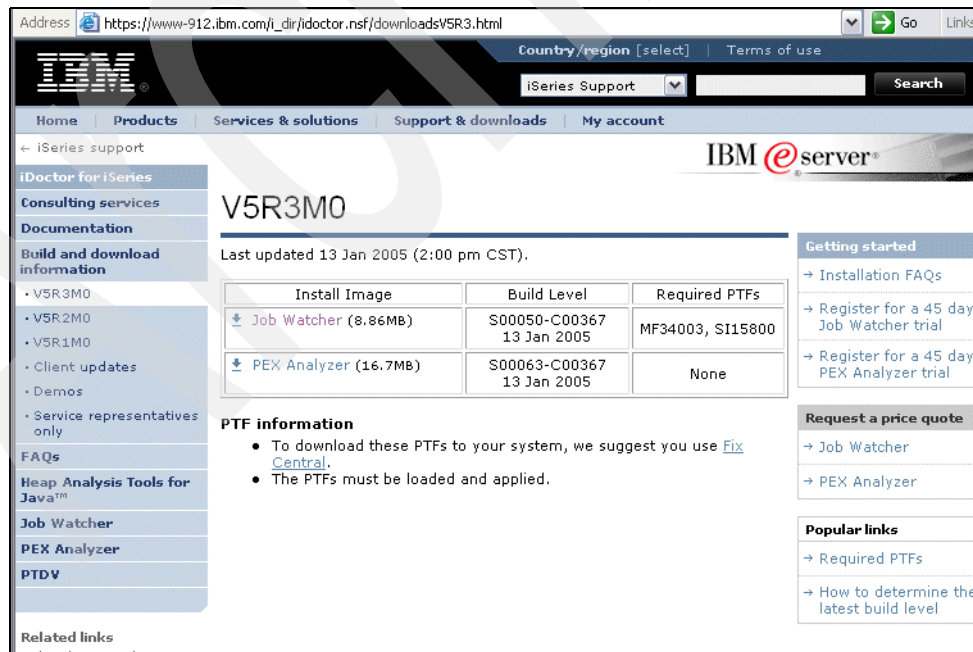


Figure 3-1 Job Watcher download Web site (March 2005)

Click the **Job Watcher** link to begin the download process. Appendix A, “Installing and uninstalling Job Watcher details” on page 191 has detailed installation instructions.

3.3 Starting a Job Watcher collection

In this section we show how to start a Job Watcher collection using the iDoctor GUI front end. The install process added an icon to your desktop that you can use to access iDoctor, or you can access the **iDoctor for iSeries** link from the Start menu.

3.3.1 Connecting to your system

1. Add a connection. If your system is not listed under the My Connections window, use the **Add Connection** option from the **File** pull-down menu to add your system to the list of available systems (Figure 3-2).

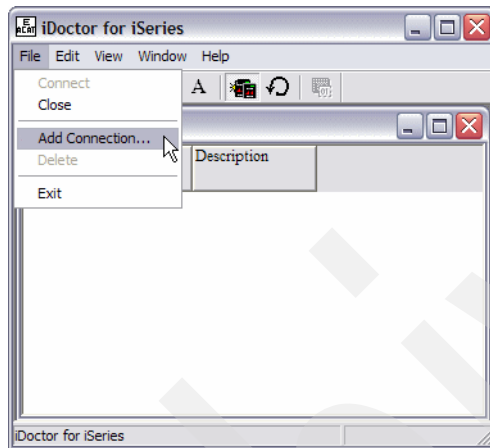


Figure 3-2 Add Connections

2. Enter your system name (Figure 3-3) and click **OK**.

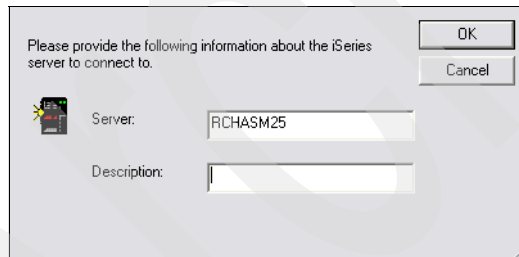


Figure 3-3 Add Connection: enter system name

3. Double-click the system name and, in the pop-up window, enter your iSeries user ID and password (Figure 3-4).

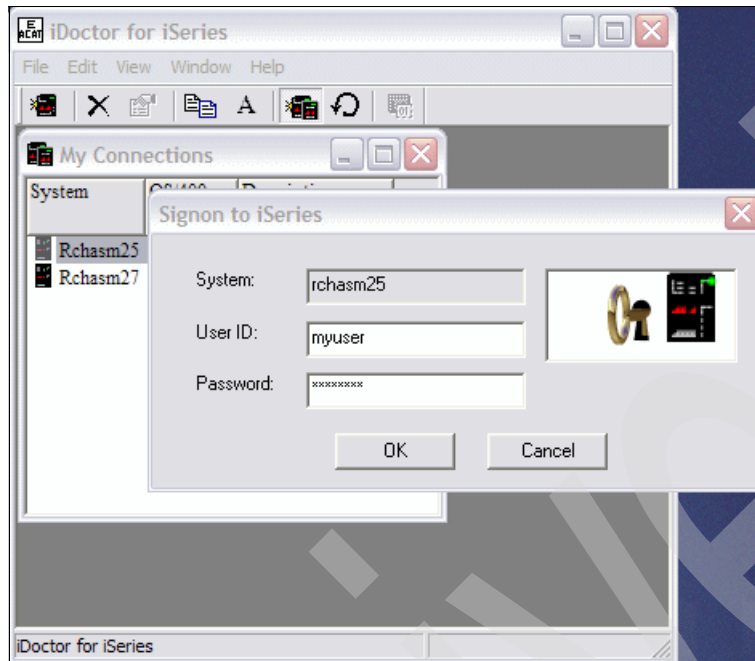


Figure 3-4 User profile verification

3.3.2 Access Job Watcher

After your user profile and password are validated, the iDoctor application panel displays a list of all iDoctor components that are installed on your system.

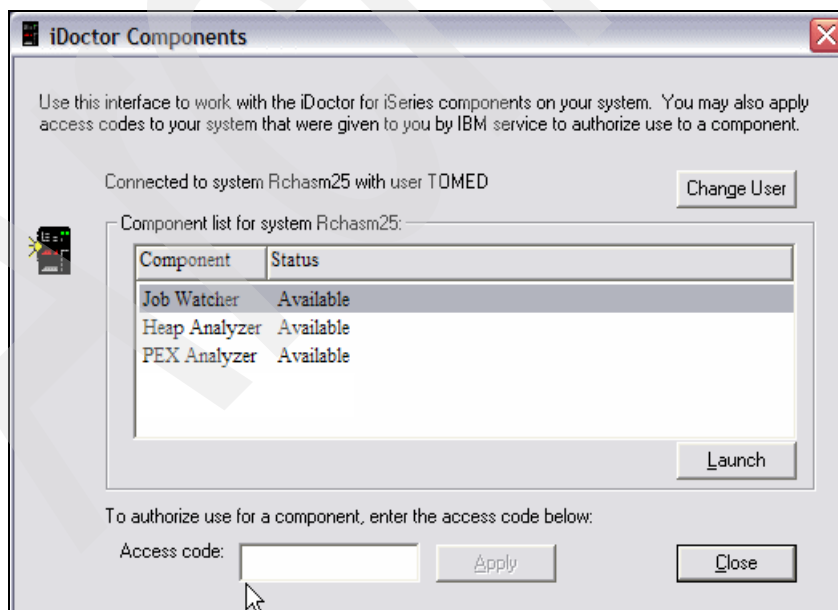


Figure 3-5 iDoctor Application component list

If you added the Job Watcher access code during the installation, the status will display as Available and you can either double-click **Job Watcher** or highlight Job Watcher and select **Launch** to continue.

If a status of error shows for Job Watcher, it probably is the result of not applying either the required PTFs or the access code. If you need to add the access code, enter it in the window and click **Apply** to make Job Watcher available.

The latest list of required Job Watcher PTFs can be found on the Job Watcher download site:

https://www-912.ibm.com/i_dir/idoctor.nsf/downloadsV5R3.html

3.3.3 Starting the Job Watcher collection wizard

The following steps show how to use the Job Watcher collection wizard to start a Job Watcher collection. If this is the first time Job Watcher has been started and there is no existing Job Watcher data on the system, skip to step 2 on page 57 to go directly to the wizard.

1. When there is existing Job Watcher data on the system, right-click **Job Watcher** in the left pane and select **Start Job Watch** (Figure 3-6).

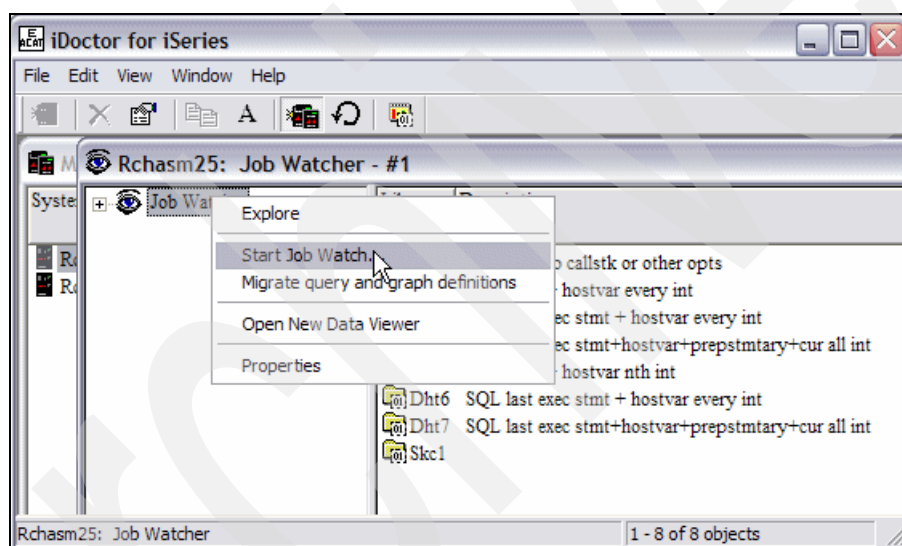


Figure 3-6 Start Job Watch

2. The Job Watcher Wizard welcome window (Figure 3-7) opens. Click **Next** to define your Job Watcher startup options and start the collection process.

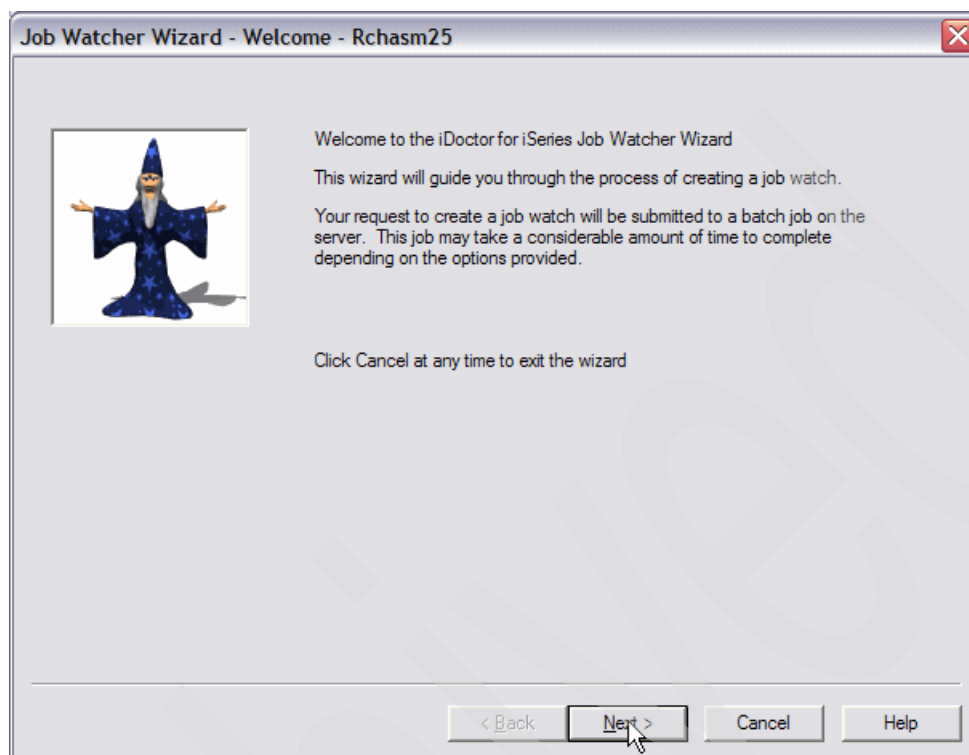


Figure 3-7 Job Watcher Wizard

3. In the Startup Options window (Figure 3-7) we define the basic startup options:
 - **Definition:** A definition is a stored set of parameter settings that can quickly be reused to set up and run multiple job watches that use the same startup options. To create a definition, enter a name in the window and select **Save As**. To use an existing definition, select the desired definition name from the pull-down menu. The startup options from the selected definition are loaded into the wizard.
 - **Job Watch Name:** This is a 10-character collection name that will be used as the file member name when creating each member in the Job Watcher output files.
 - **Library:** This is the name of the library on the iSeries where your Job Watcher collection output files will be created. If the library does not already exist, Job Watcher asks whether you want to create the library now. If you decide not to create the library, you must specify a library that already exists.
 - **Interval duration:** The approximate elapsed time in seconds between each data collection sample. On a very busy system it is possible for the Job Watcher collection job to take slightly longer than the specified interval duration to complete (1/10 second longer has been observed).

There is also the option to collect the data as fast as possible. If the Collect as fast as possible box is checked, the interval duration parameter will be grayed out.

Note: Job Watcher can collect a large amount of data in a very short period of time when collecting as fast as possible.

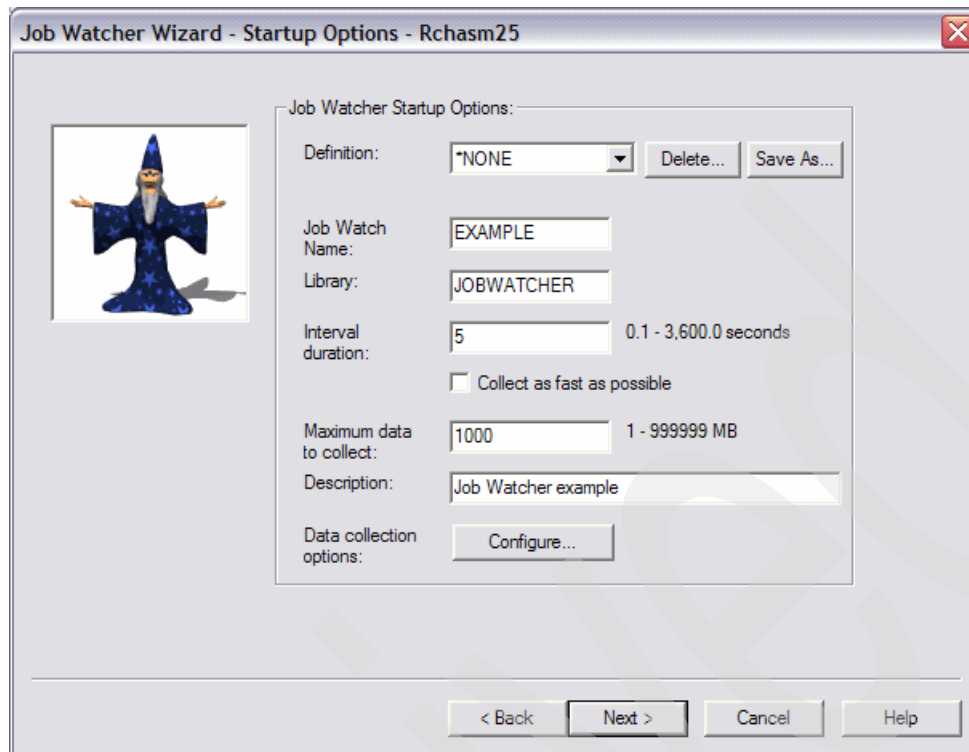


Figure 3-8 Startup Options

Tip: When trying to determine the programs or procedures that are causing high CPU usage in a specific job, set the collection interval to as fast as possible and have the collection run for a duration of no more than five minutes. This will produce a more accurate view of what programs or procedures are using CPU.

- **Maximum data to collect:** This is the maximum size that you allow the Job Watcher output database to grow in megabytes. Valid options are 1 to 999,999. If you happen to enter a value larger than the space available on your system, Job Watcher will stop automatically when your DASD threshold is reached. We go into more detail on stopping Job Watcher in 3.4, “Stopping a Job Watcher collection” on page 78.
- **Description:** A description is optional but may be helpful if someone else will analyze this data, in which case you may want to describe why you started this collection.
- **Data collection options:** Described in the next section, “Data collection options.”

3.3.4 Data collection options

Selecting the **Configure** button next to Data collection options gives you the ability to fine tune your data sampling. Learn more about the following topics and figures in the iDoctor documentation, which you can download from:

https://www-912.ibm.com/i_dir/idoctor.nsf/documentation.html

Data collection options: call stack

By default, Job Watcher collects call stacks during every interval. See Table 3-1 on page 59 for a description of each option. In Figure 3-9 on page 59 we have selected **Advanced** so that you can see these additional options.

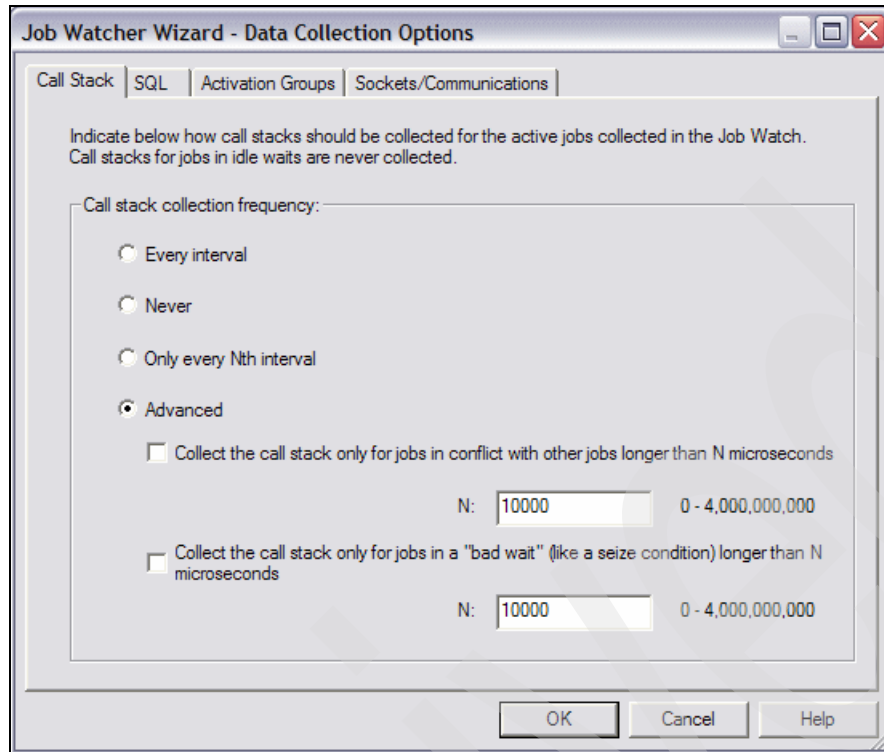


Figure 3-9 Data Collection Options: Call Stack tab

Tip: Call stack information can be useful in determining what programs or procedures are responsible for wait conditions such as objects locks and seizes. This facilitates a shorter problem resolution time.

Table 3-1 gives a description of each of the available options.

Table 3-1 Call stack options

Call stack options	Description
Every Interval	The call stack will be collected every interval for every active job in the collection. An active job is a job that used CPU.
Never	The call stack will never be collected
Only every Nth. interval	The call stack will be collected only for active jobs every Nth interval. Selecting this displays a field where the value of Nth can be entered. For example, if the value for Nth is 5, then only jobs that used CPU every 5th interval of the collection will be collected.
Advanced	Provides the option to dynamically collect call stacks based on performance statistics found in the data. Choosing this option enables you to collect call stacks, for example, for only those jobs that are in conflict or holding other jobs longer than a specified time, or only for jobs that are in a bad wait condition, such as a seize.

Data collection options: SQL

On the SQL tab, the default is Do not collect any SQL statements. In Figure 3-10, we selected to collect Active SQL statements and host variables. Chapter 5, “SQL, call stack, and journal analysis examples” on page 125, shows SQL statement data collection examples.

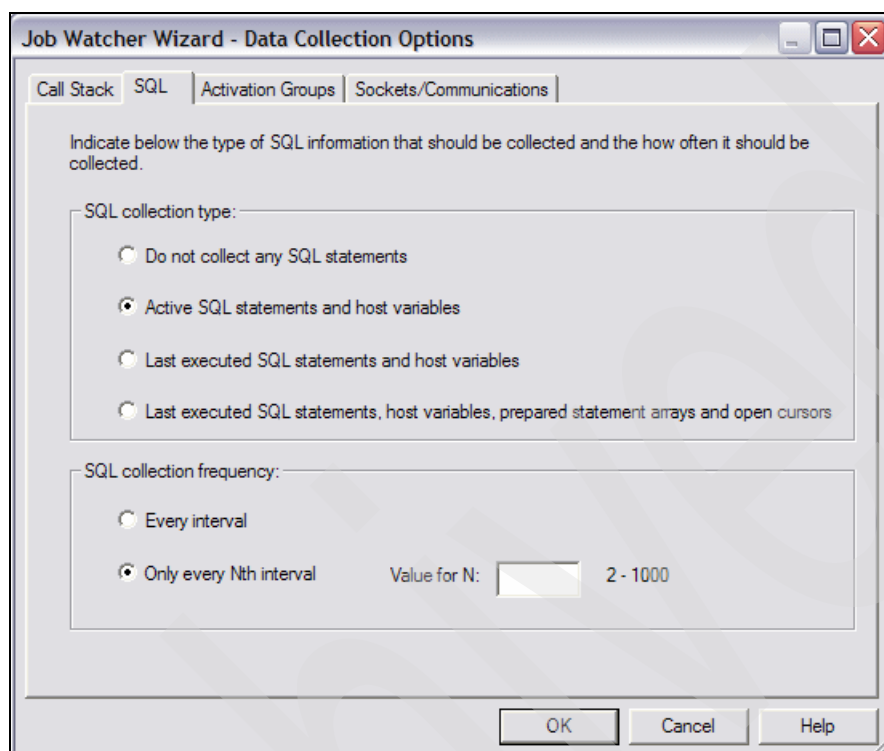


Figure 3-10 Data Collection Options: SQL

Table 3-2 gives a description of each of the available SQL options.

Table 3-2 SQL options

SQL options	Description
Do not collect any SQL statements	No SQL statements will be collected.
Active SQL statements and host variables	SQL statements and host variables will be collected for any job that is currently running SQL statements within the collection.
Last executed SQL statements and host variables	Collects the last executed SQL statement and host variables for every job in the collection, for every interval the job is active.
Last executed SQL statements, host variables, and prepared statement arrays	Collects the last executed SQL statement and host variables for every job in the collection, for every interval the job is active. It also collects information about prepared statement arrays and open cursors for the job running the SQL statement.
SQL collection frequency	This displays only if you select one of the options to collect SQL statements. You can collect every interval or only every Nth interval.

Tip: Active SQL statements and host variables are used when analyzing a performance issue where we are unsure if SQL is being used. The collected data shows any jobs using SQL, and if they are long running SQL statements they appear across multiple intervals.

Data collection options: Activation groups

Figure 3-11 shows the Activation Groups tab. The default is to not collect any activation group information. For a complete description of each option, see Table 3-3.

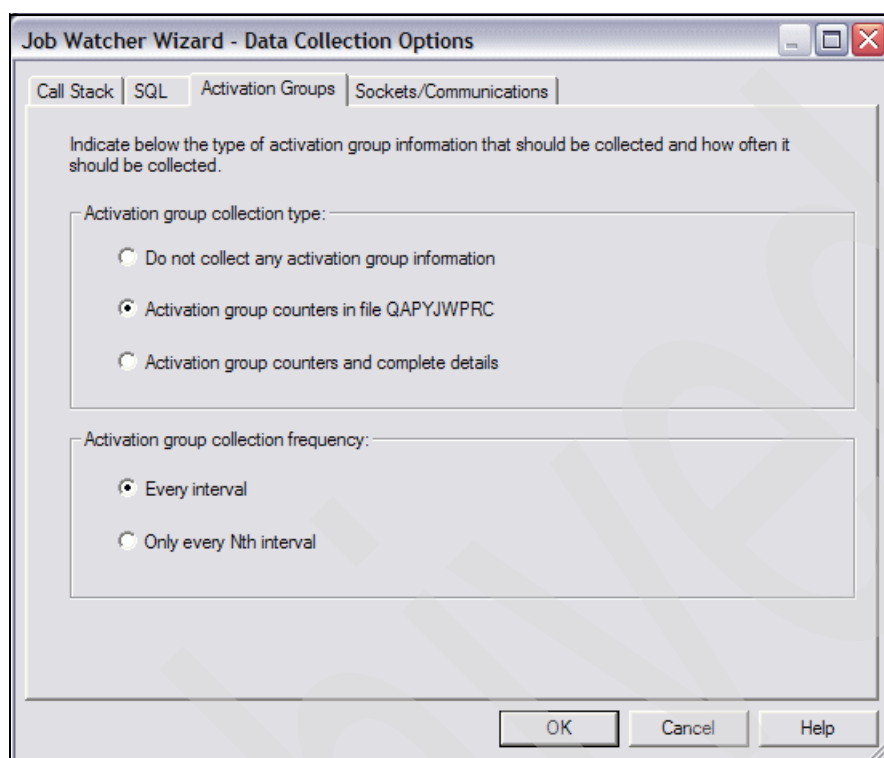


Figure 3-11 Data Collection Options: Activation Groups tab

Tip: Heavy activation group create and delete activity is very expensive, and a significant increase can indicate a problem such as a new level of software with some non-optimal performance design choices.

Table 3-3 lists a description for each of the available options.

Table 3-3 Activation group options

Activation group options	Description
Do not collect any activation group information	No activation group data will be collected.
Activation group counters in file QAPYJWPRC	If this option is selected, counters in the qapyjwprc file (the job or process information file) will be filled. The fields that will be filled are curnumactg (current number of activation groups) and curnumact (current number of activations).
Activation group counters and complete details	Collects the activation group counters in the prior option as well as additional files containing complete information about the activation groups for all jobs in the Job Watch: qapyjwaigp - general activation group information qapyjwaihp - activation group heap sizes and counts qapyjwaipa - list of programs in each activation group collection
Activation group collection frequency	This is available only if you select one of the options to collect activation group data. You can collect every interval or only every Nth interval.

Socket communication data collection options

On the Sockets/Communications tab (Figure 3-12), the default is to never collect socket communications data. Table 3-4 lists a description for each option.

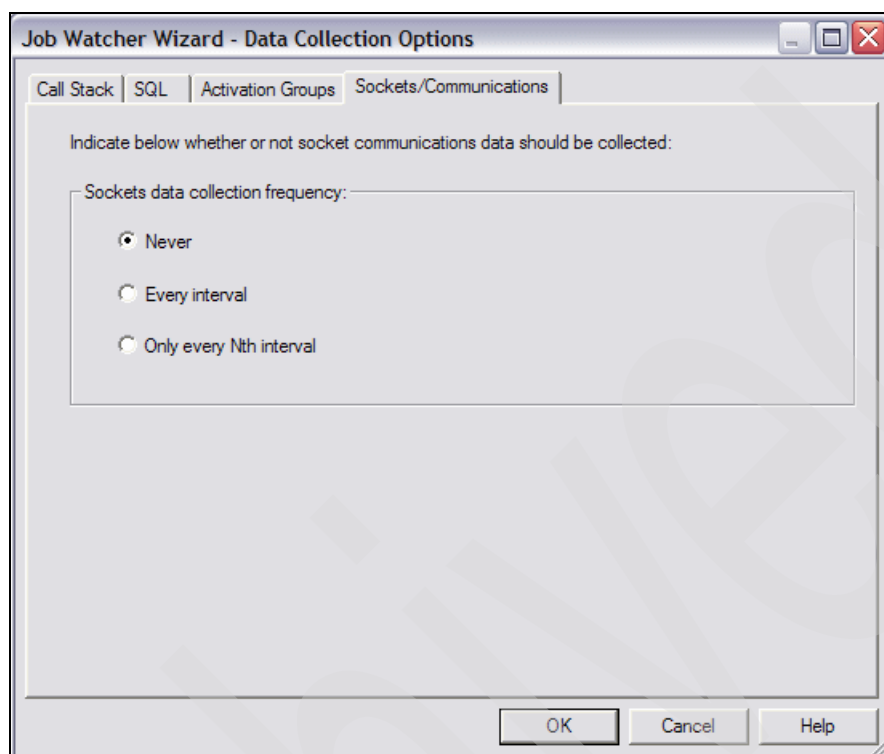


Figure 3-12 Data Collection Options: Sockets/Communications tab

Table 3-4 is a description of each of the available options.

Table 3-4 Sockets/Communication data collection options

Option	Description
Never	Never collect socket or communication data. Never is currently the default option.
Every Interval	Collect socket and communication data every interval
Only every Nth interval	The socket and communication data will be collected only for active jobs every Nth interval. Selecting this field will display a field where the value of Nth can be entered. For example, if the value for Nth is 5 then only every 5th interval of the collection will be collected.

3.3.5 Job and task options

In this section we define what jobs and tasks we want to collect in our Job Watcher collection.

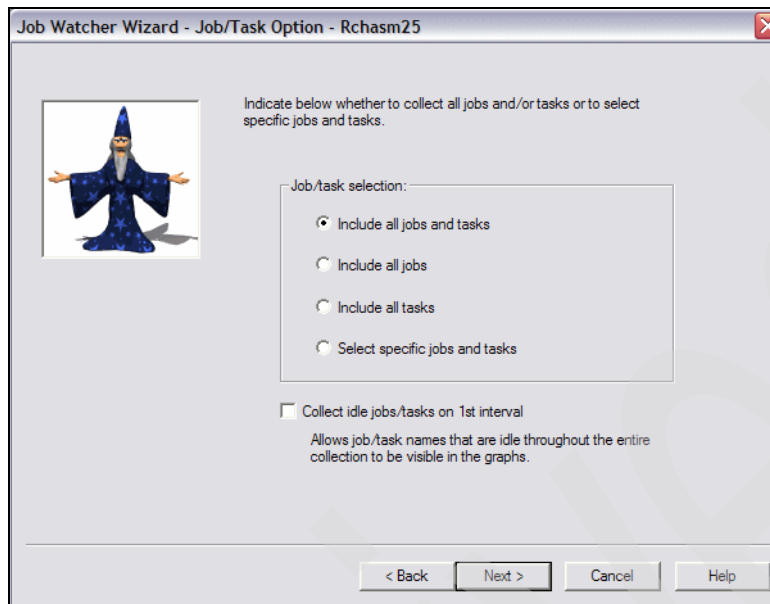


Figure 3-13 Job/Task Options window

Table 3-5 has a description of each option.

Table 3-5 Jobs/Tasks options

Option	Description
Include all jobs and tasks	This option includes all user and system jobs as well as all system tasks. Prior to V5R3 this was called a <i>system-wide</i> collection.
Include all jobs	This option includes all user and system jobs.
Include all tasks	This option includes all system tasks.
Select specific jobs and tasks	Selecting this option provides additional panels where we can specify specific jobs or tasks that you want to include in your collection. See “More on selecting specific jobs and tasks” on page 64.
Collect idle jobs on first interval	Selecting this option forces all jobs and tasks to be included in the first interval. Normally only jobs and tasks that use CPU during an interval would be included. If starting Job Watcher when a problem is already in progress, it is a good idea to include idle jobs and tasks. If starting Job Watcher prior to a problem occurring there is no need to include idle jobs and tasks.

3.3.6 More on selecting specific jobs and tasks

Currently there are six different methods you can use to select jobs and tasks that you want to include in your Job Watcher collection. Use the pull-down menu next to **Select by** to select the method you want to use (Figure 3-14).

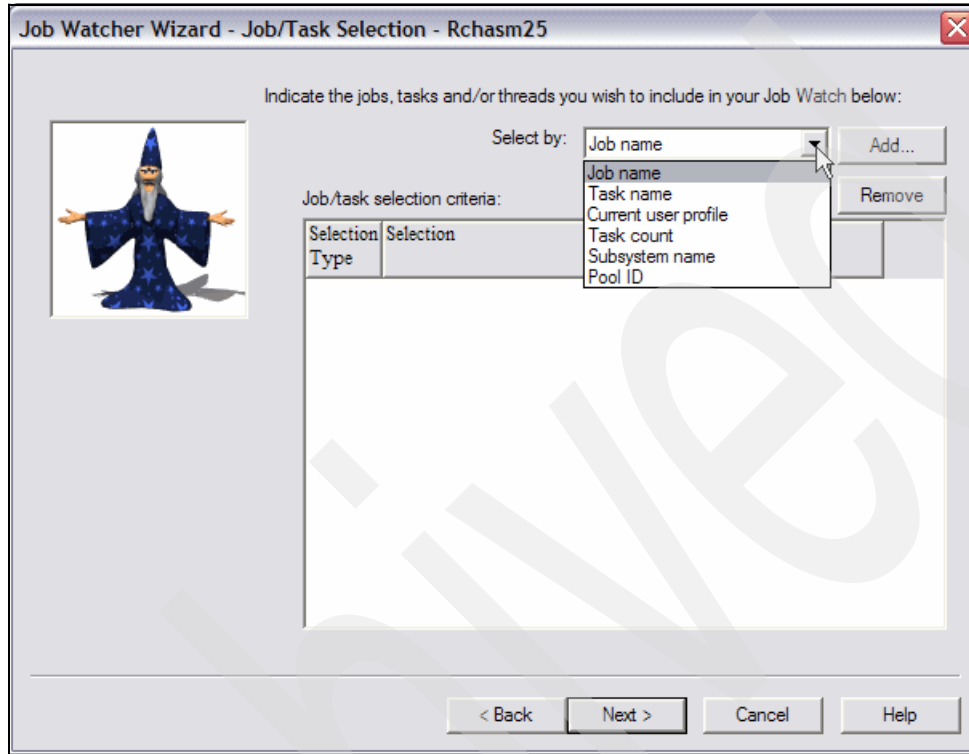


Figure 3-14 Job/Task Selection

In this section we cover these six methods and explain how to use them.

Job name

Clicking **Add** displays a panel that enables you to create a list of jobs by using filters. You can filter the jobs you want to display by using a combination of the following; *jobname*, *job number*, *user profile*, and *current user*.

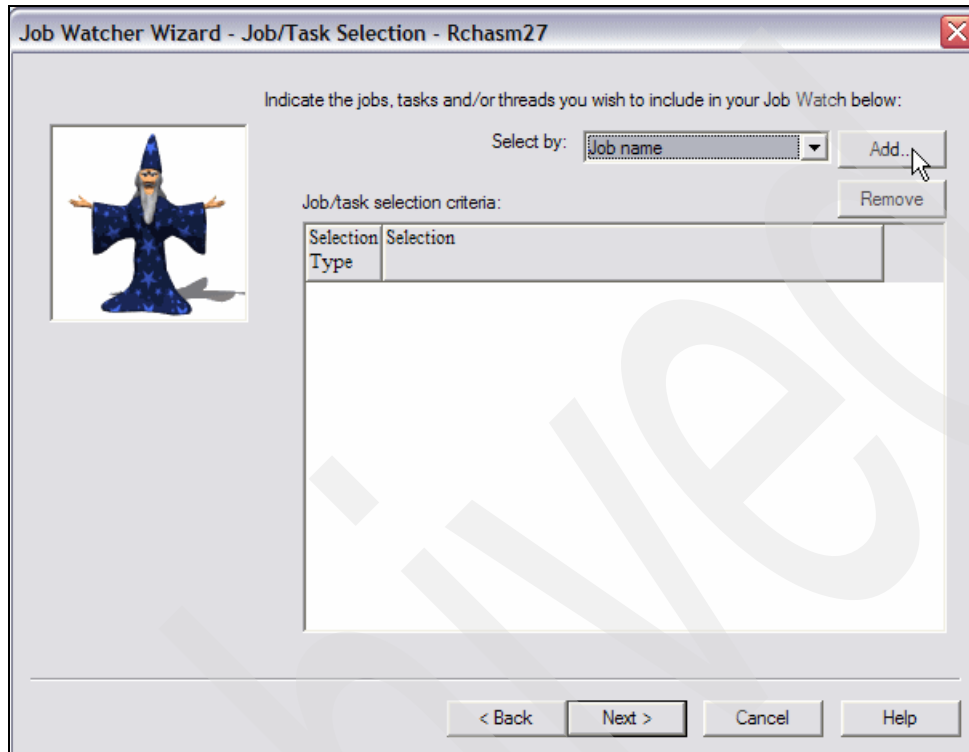


Figure 3-15 Job/Task selections: Job name

1. In Figure 3-16 we specify the user tomed. By clicking **Refresh**, we get a list of all jobs that user tomed has active. By using the pull-down menu for the Status option, we can also change the status to **Jobq**. This lists jobs that user tomed has in jobq status.

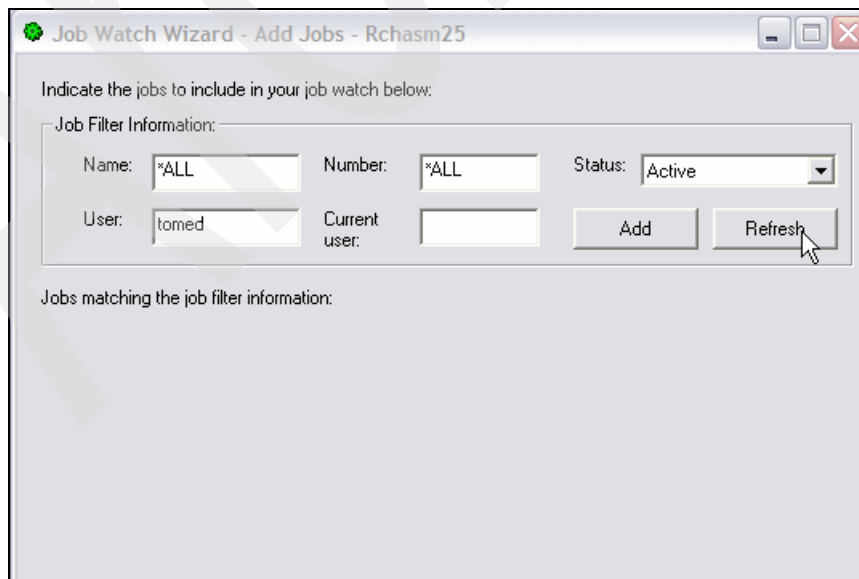


Figure 3-16 Add job list

2. By highlighting one or more jobs and clicking **Add Selected**, we add these jobs to our collection. Click **Close** to continue.

Job Watch Wizard - Add Jobs - Rchasm25

Indicate the jobs to include in your job watch below:

Job Filter Information:

Name: Number:

User: Current user:

Jobs matching the job filter information:

Subsystem	Job Name	User	Number	Function	Current User	Entered System On
QBATCH	LOOPER1	TOMED	003262	STRLOOPER3	TOMED	10/11/04 10:41:58
QINTER	QPADEV0004	TOMED	003259	WRKOBJPDM	TOMED	10/11/04 10:39:35
QINTER	QPADEV0005	TOMED	003260	CMDENT	TOMED	10/11/04 10:41:12
QBATCH	STRLOOPER	TOMED	003261	60	TOMED	10/11/04 10:41:59

Figure 3-17 Add jobs

3. This returns us to the Job/Task Selection window (Figure 3-18) where we click **Next** to continue setting up our Job Watcher collection.

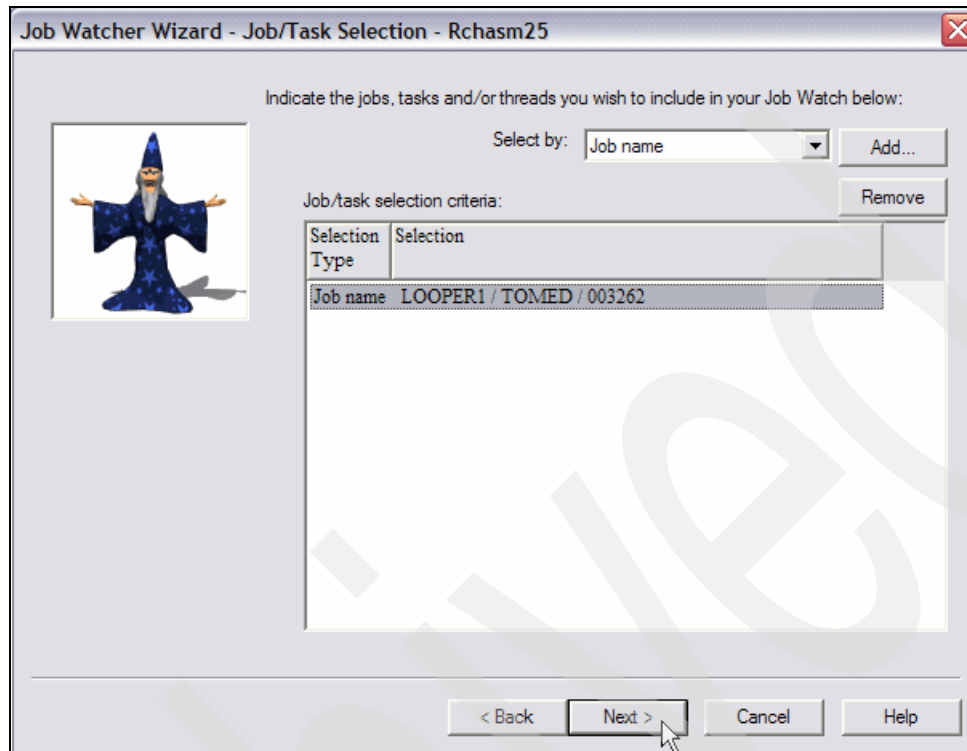


Figure 3-18 Job/Task Selection

Tip: With the **Remove** button, you can remove any selection that you decide not to include.

Task name

The task name would be a valid system task. Although we cannot build a list of active tasks as we could when selecting a job, we can use generic names. For example, include SMP* for all tasks starting with SMP and this would include the storage management page out tasks.

Using the **Select by** pull-down menu, choose **Task name** and click **Add** to display a window for entering the tasks that you want to include (Figure 3-19).

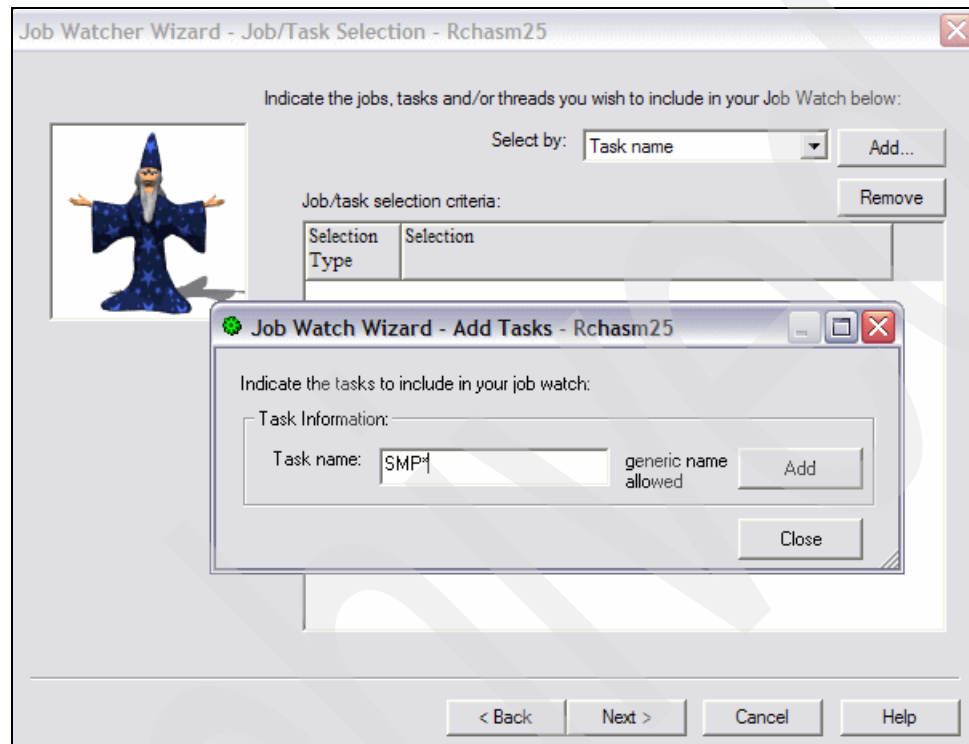


Figure 3-19 Add system task name

You can include multiple system tasks by entering each task name or generic task name individually and clicking **Add**. After you have entered all system tasks that you want to include in your Job Watcher collection, click **Close**, then click **Next** to continue.

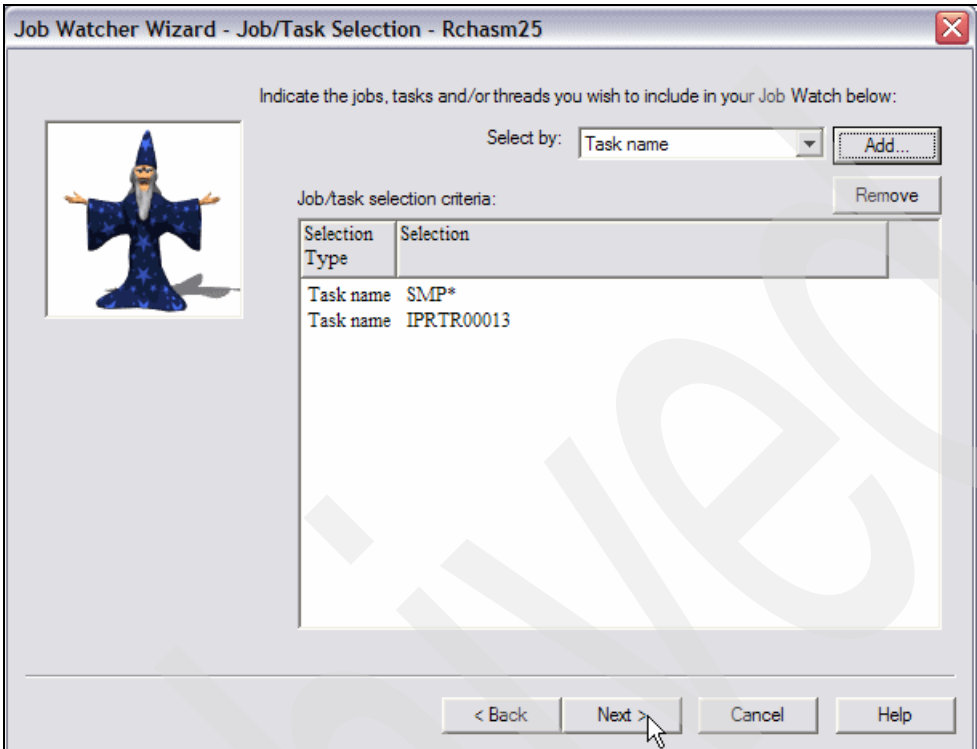


Figure 3-20 Job/Task selection

Current user profile

This option is very useful when you want to include jobs in your Job Watcher collection that run under a generic user profile but service specific users. For example, the QZDASOINIT jobs are submitted by QUSER. If you display the job and look at the job status attributes you will see the job’s current user profile, which is the user that is being serviced by this job. In the example in Figure 3-21, this QZDASOINIT job is servicing the current user TOMED.

Display Job Status Attributes			
Job:	QZDASOINIT	User:	QUSER
		Number:	005281
Status of job	:	ACTIVE
Current user profile	:	TOMED
Job user identity	:	TOMED
Set by	:	*DEFAULT
Entered system:			
Date	:	10/19/04
Time	:	16:56:37

Figure 3-21 Job status attributes

From the **Select by** pull-down menu, choose **Current user profile** and click **Add** to enter the user profile that you want to include in your Job Watcher collection (Figure 3-22).

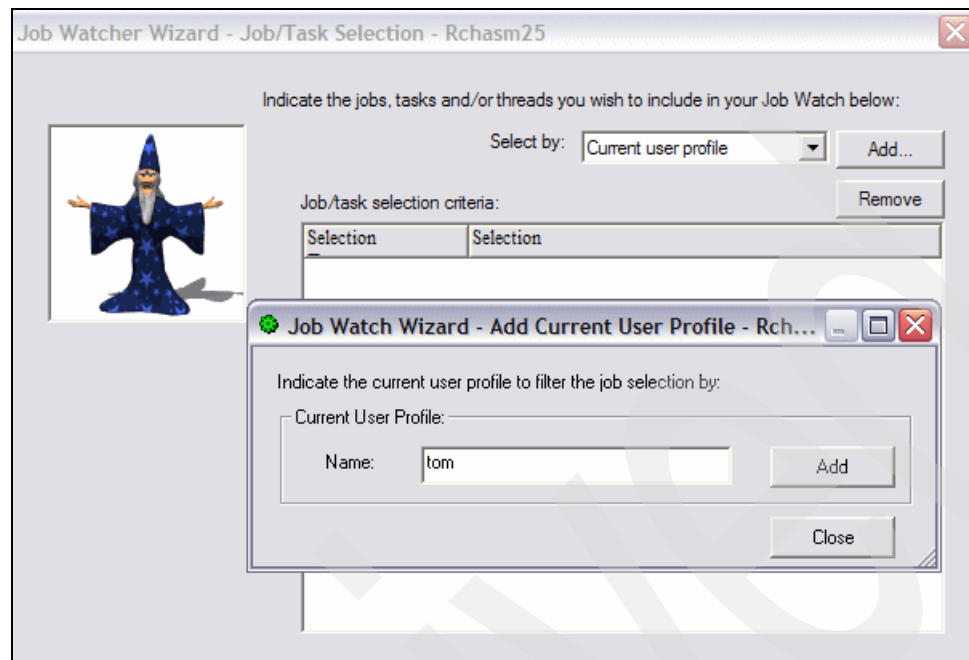


Figure 3-22 Add current user profile

Click **Close** to return to the Job/Task Selection window. After you have added all current user profiles that you want to include in your Job Watcher collection, click **Next**.

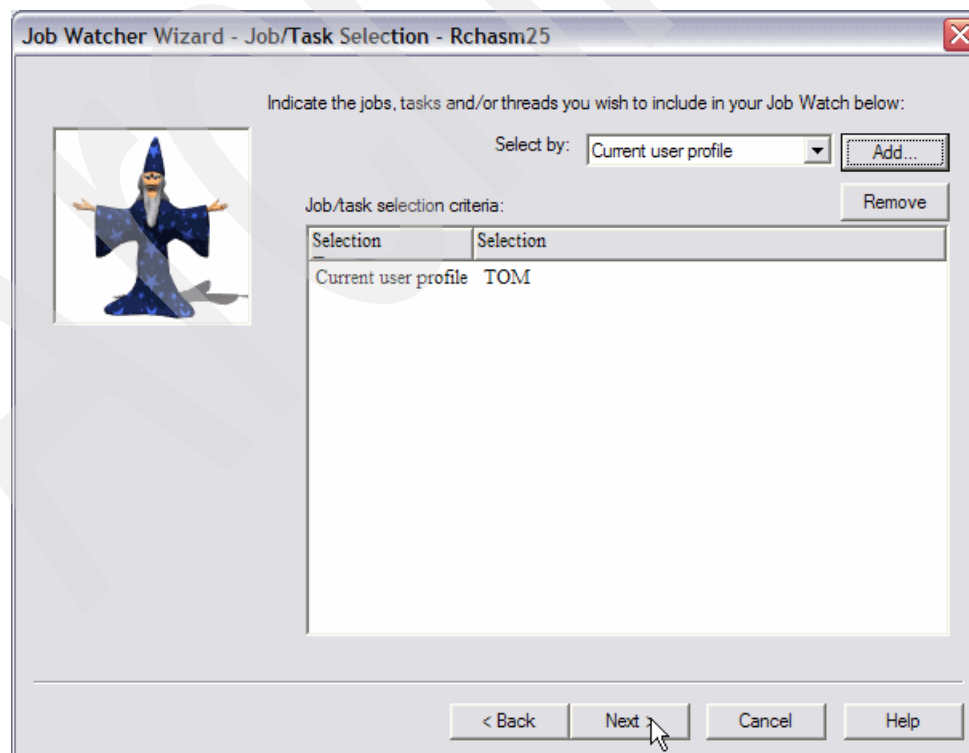


Figure 3-23 Job/Task selection

Task count

Task count is a unique value that the Licensed Internal Code (LIC) layer of the system assigns to every dispatchable unit of work (job, thread, or task), as described in more detail in 2.2, “Job Watcher terminology” on page 23.

Use the **Select by** pull-down menu to choose **Task count** and click **Add** to enter the task count that you want to include in your Job Watcher collection, as shown in Figure 3-24.

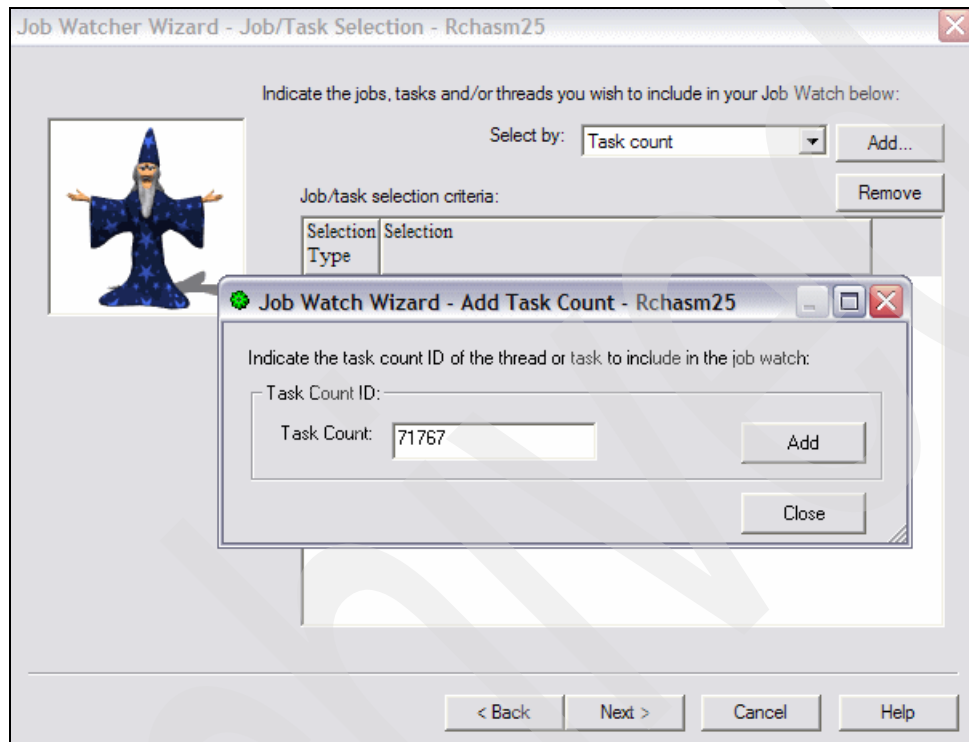


Figure 3-24 Add Task Count

Tip: Other than using the system service tools (for example, with the Start Service Tools command, STRSST) or a Performance Explorer trace, the only way to determine a job’s task count is from an existing Job Watcher collection. Therefore we would use this option only as a result of analyzing either a currently running, system-wide Job Watcher collection or a Job Watcher collection that included a large set of jobs and we found a need to harvest more detailed information about a specific job that was exhibiting unusual wait or runtime conditions.

Subsystem

Using the subsystem option enables you to define a Job Watcher collection for all jobs that are running under a specific subsystem or group of subsystems.

From the Add Subsystem window (Figure 3-25), use the pull-down menu to select the subsystems that you want to include by highlighting a subsystem name and clicking **Add**. Repeat for each additional subsystem you want to add.

You can also specify non-existent subsystem descriptions. If such a subsystem becomes active later, Job Watcher will collect data for the jobs or threads in that subsystem if they become active. When you have finished adding all of the subsystem names, click **Close** to return to the Job/Task Selection window.

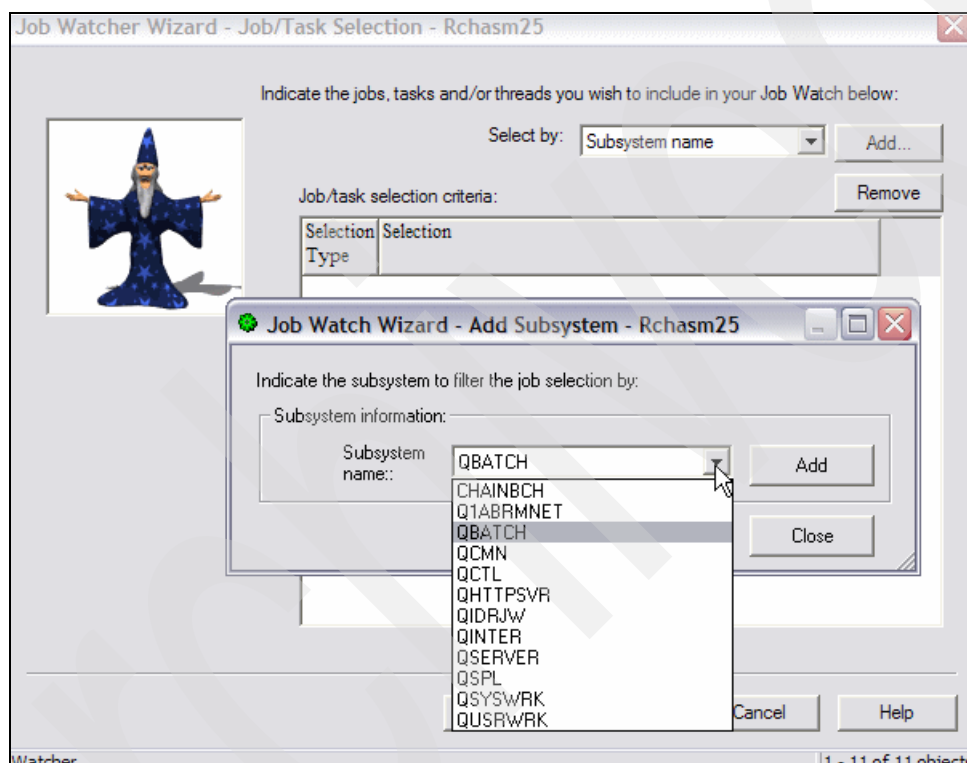


Figure 3-25 Add Subsystem selection

Pool ID

The final selection method we examine is pool ID. If you suspect that the faulting rate in a pool is too high this is a good method to use to determine the exact cost these faults are having on the jobs running in this pool.

Use the **Select by** pull-down menu to choose **Pool ID**, and select **Add** to display the window where you enter the pool ID you want to include in your Job Watcher collection. Job Watcher will collect information for jobs and threads running in that pool when that pool is actually being used.

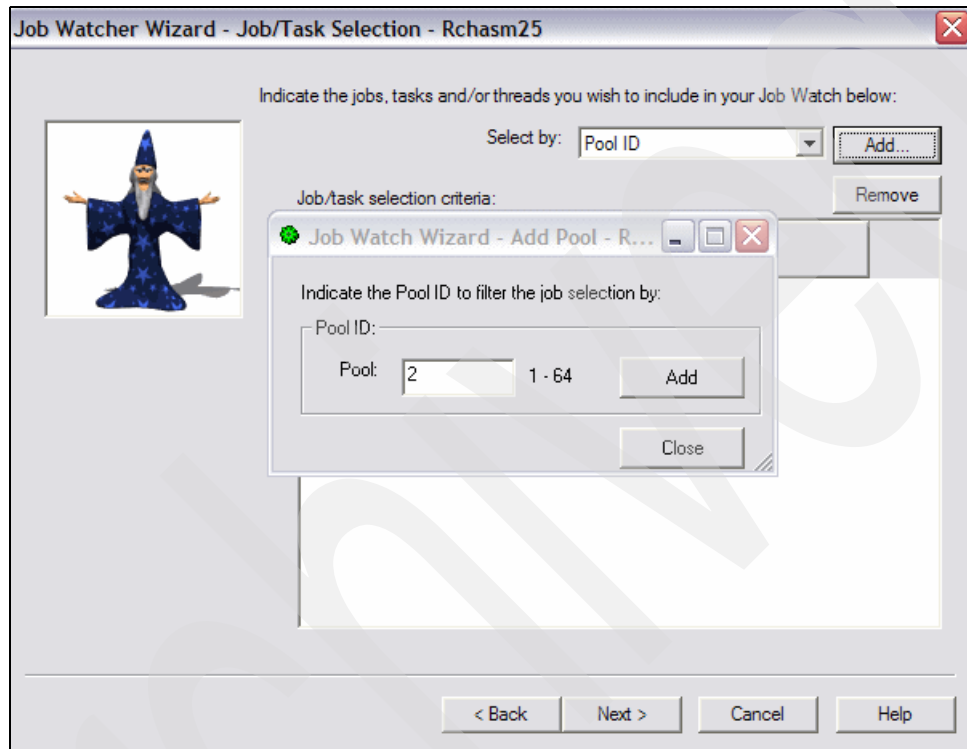


Figure 3-26 Select Pool ID

Job Watcher validates that you have entered a value between 1 and 64 but does not check to ensure that the value you entered is a valid system pool ID. That is, if one of the specified pool IDs is not active during the Job Watcher collection period, it is merely not recorded in any Job Watcher data.

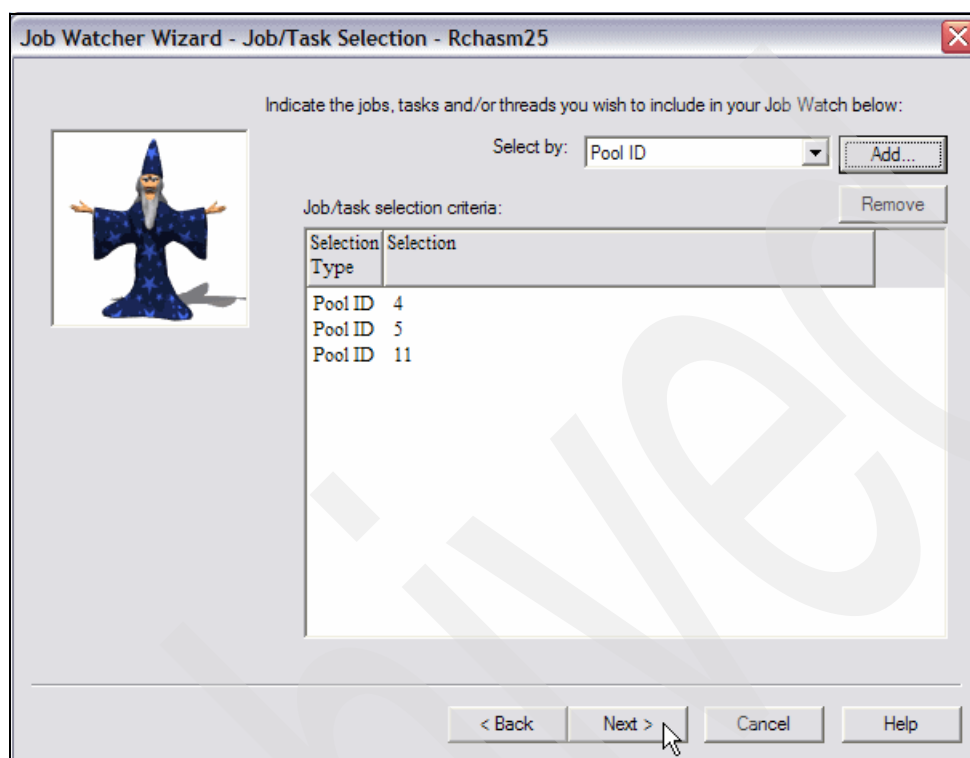


Figure 3-27 Pool ID selection

After you have entered all of the pool IDs that you want to include in your Job Watcher collection, click **Next**. This displays options for ending the Job Watcher collection.

Important: We have briefly shown you several different options so that your collection does not collect information about every started job, thread, or task during a Job Watcher collection. Even with these options, you may still record many thousands of jobs, threads, and tasks on a very busy system. A typical example of this would be systems or partitions with thousands of HTTP server users or Domino mail users.

In those cases with perhaps 90,000 threads (most not consuming CPU) that are available for viewing, keep in mind the content of 3.5, “Viewing Job Watcher data” on page 78. You should look at the graphs showing jobs, threads, or tasks ranked by whatever statistics that you decide are the most interesting to you, such as graphs ranking jobs, threads, or tasks by CPU, by seizures, or even by I/O rates, or by access to the integrated file systems (IFS), 5250 transactions, and so on.

3.3.7 Ending Options window

The Ending Options window is where we define how long we want our Job Watcher collection to run. As we see in the following examples, two options will determine how long our collection will run:

- **Collect for a specific number of intervals:** Use this option to specify how many intervals you want Job Watcher to collect. Valid options are 1 through 2,000,000,000. This also

gives the estimated collection run time based on the number of intervals and the interval length. See the example in Figure 3-28.

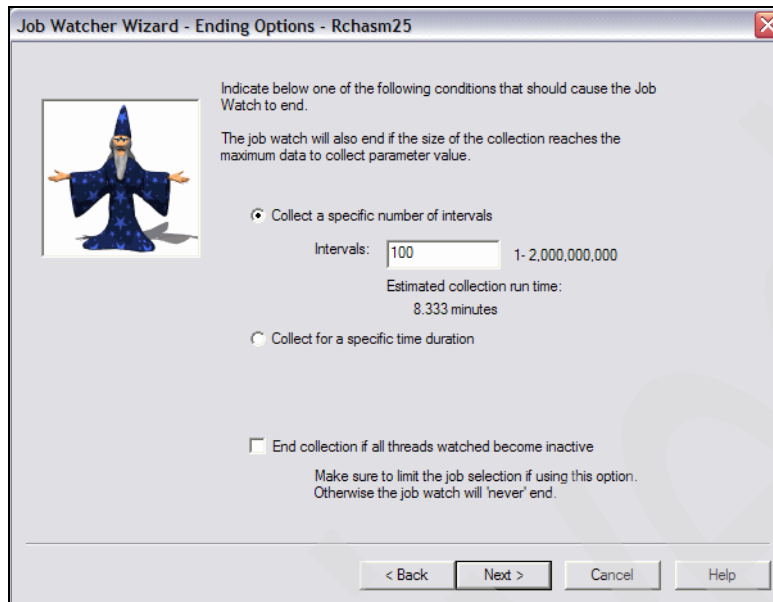


Figure 3-28 Ending Options: Intervals

- **Collect for a specific time duration:** When selecting this option, a field is displayed in which you can specify the time in seconds, minutes, hours, or days. The default is seconds and the value can contain a decimal value. See the example in Figure 3-29.

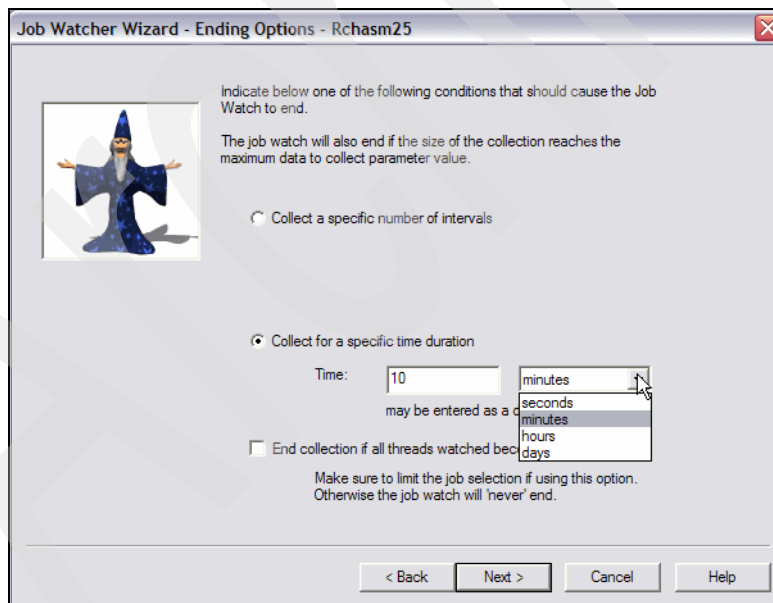


Figure 3-29 Ending Options: Time

- **End collection if all threads watched become inactive:** This option is available only when select jobs and tasks are specified. Using this option causes Job Watcher to end when all jobs and tasks that are included in the collection have ended.

3.3.8 Job Watcher Wizard summary

The last window we see before our Job Watcher collection starts is a summary of the options that we have selected for our Job Watcher collection.

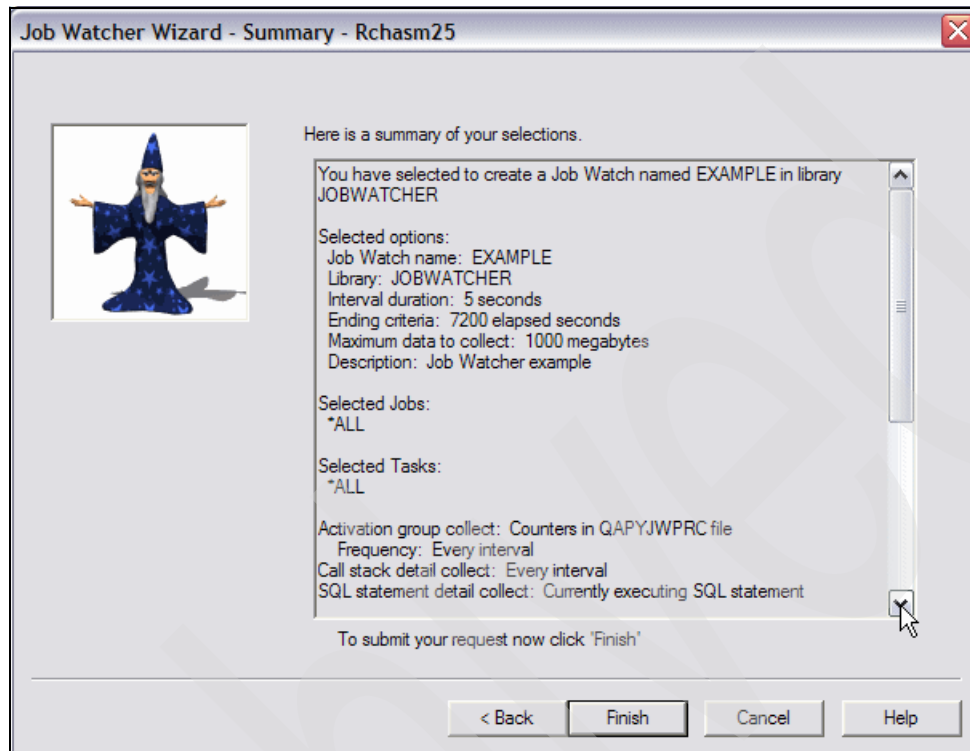


Figure 3-30 Job Watcher options summary

Scroll down the Summary window to see the actual command that is submitted to the iSeries or i5 system (Figure 3-31 on page 77). The remote command string shown here can also be copied and pasted to a 5250 command line or to a CL program source for use in other operational environments.

Click **Finish** to submit the WCHJOB command and start our Job Watcher collection. The Job Watcher collection job runs in subsystem QSYS/QIDRJW.

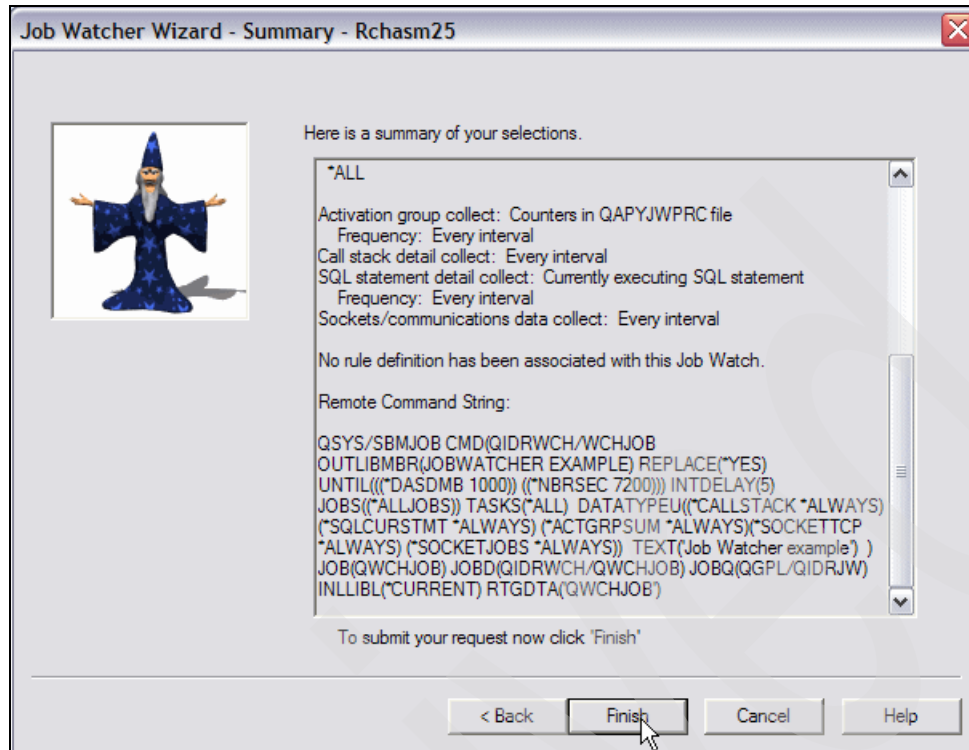


Figure 3-31 Job Watcher summary options continued

3.3.9 Active Job Watcher

After you have submitted your Job Watcher collection, the iDoctor for iSeries window opens (Figure 3-32). You may have to expand **Job Watcher** in the left pane to display the list of libraries that contain Job Watcher data. From the library list, select the library that you used for your Job Watcher data. This displays your collection in the right pane; in the example in Figure 3-32 you can see that this collection is still in progress, which means that our collection is still active.

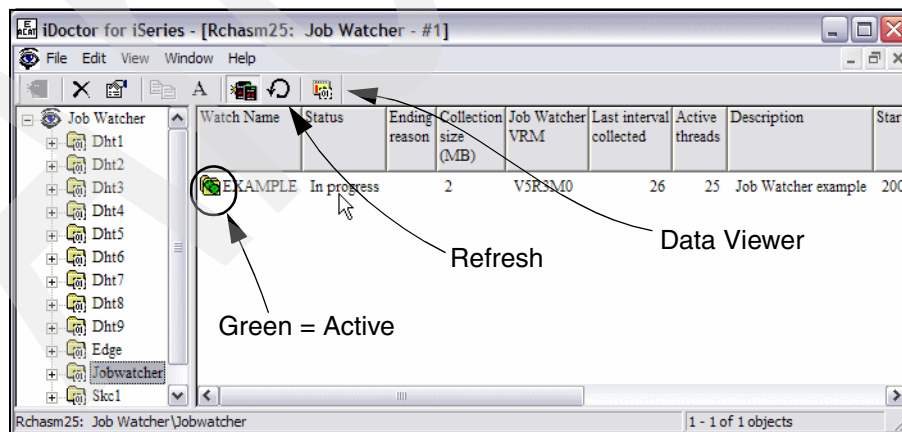


Figure 3-32 Active Job Watcher example

As well as showing the status of our collection we can also see the size, release level, and the last interval collected. For a collection that is in progress, the last interval value will increase with each refresh of the window. We are also shown the number of active threads and

description, and if we scroll to the right, we see the time that we started the collection and the time the last interval was collected.

Tip: Next to a collection's *watch name* is a folder (icon) with two gears on it. When these gears are green as we see to the left of watch name EXAMPLE, the collection is still active. When the gears change to red, the collection has completed and is ready for analysis. We do not have to wait for a collection to complete; we can also view graphs and analyze a collection up to the most recently completed collection interval while active as well.

Job Watcher collection functions are performed by server job QPYSWJOB running in subsystem QIDRJW. Its collected data is stored in multiple i5/OS Job Watcher database files in the library you specified when starting the collection. Your Job Watch name (collection name) is used as the member name in each of these files to associate the collected data to a specific collection.

The Job Watcher client Graphical User Interface (GUI) component performs its graphical views of the collected data by real time access to the iSeries server's Job Watcher database file members.

3.4 Stopping a Job Watcher collection

The following are the different ways that a Job Watcher collection can or will stop:

1. Time limit reached: As we saw in the previous section we can specify a time limit to the collection run time. See 3.3.7, "Ending Options window" on page 74 for more details about setting a time limit.
2. Maximum intervals reached: We also covered this in the previous section, where we were able to set the number of intervals that we wanted to collect data for. See 3.3.7, "Ending Options window" on page 74 for more detail on setting the intervals.
3. Maximum data collected: We set this value when we defined our start-up options; this is the maximum size in megabytes that we are allowing the Job Watcher database to grow.
4. Auxiliary storage threshold reached: Starting with V5R3, Job Watcher will stop automatically if the system auxiliary storage threshold is reached. This will prevent system crashes as a result of auxiliary storage reaching 100%.
5. Termination of the job QWCHJOB: If the Job Watcher collection job is terminated, the collection of data will also stop. But the collection data will still be valid up to the last interval that was written to the output files.
6. Using the stop option: There is also an option via the iDoctor GUI application to stop your collection by right-clicking your collection and selecting **Stop**.
7. All threads of the watched jobs terminate, thereby satisfying the "stop when all threads end" start option.

3.5 Viewing Job Watcher data

Job Watcher collections are analyzed using the iDoctor client workstation GUI component. Throughout the remainder of this book, this component will be referred to simply as the *Job Watcher GUI*, *GUI*, or the *iDoctor GUI*, unless otherwise noted. If the iDoctor GUI is not already open, click the iDoctor for iSeries desktop icon, connect to your system, and launch Job Watcher. If you need help starting Job Watcher, see 3.3.1, "Connecting to your system" on page 54 and 3.3.2, "Access Job Watcher" on page 55 for more information.

3.5.1 Job Watcher view

Job Watcher uses a split window view, with a tree pane on the left side and a list on the right. The initial list that displays when launching Job Watcher is a list of libraries on the system containing Job Watcher output data. By right-clicking on a Job Watcher library, you can choose the **Select fields** option, which enables you to change the layout of the fields that are displayed in the Watch Name list (Figure 3-33).

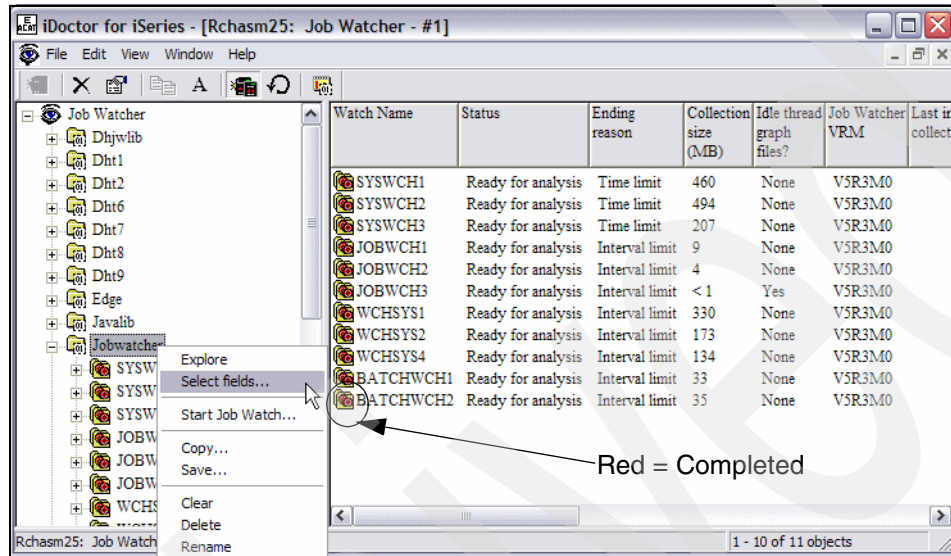


Figure 3-33 Select fields

The Select fields option displays a window that enables you to remove fields and change the order in which the fields are displayed. Move a field up or down the list by clicking on and dragging the field to the desired location within the list. Figure 3-34 shows an example of moving the start date to below the collection status field. A field can be removed from being displayed by removing the check mark next to the field name.

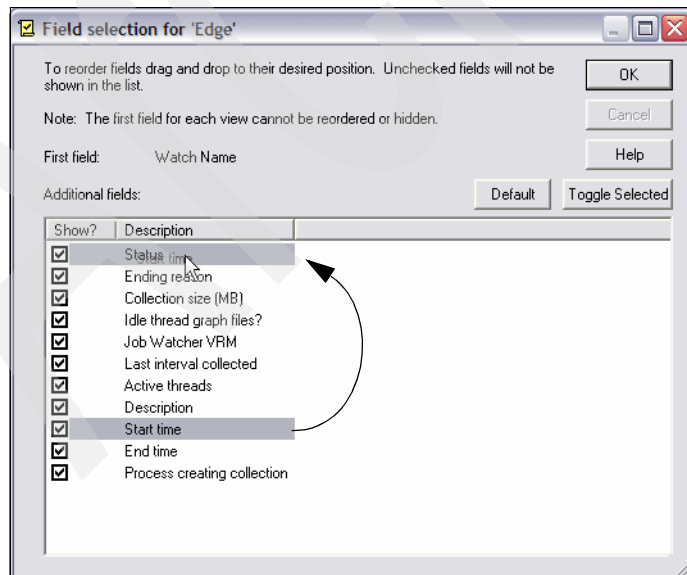


Figure 3-34 Field selection for Job Watcher

Figure 3-35 on page 80 shows what the window looks like after our changes.

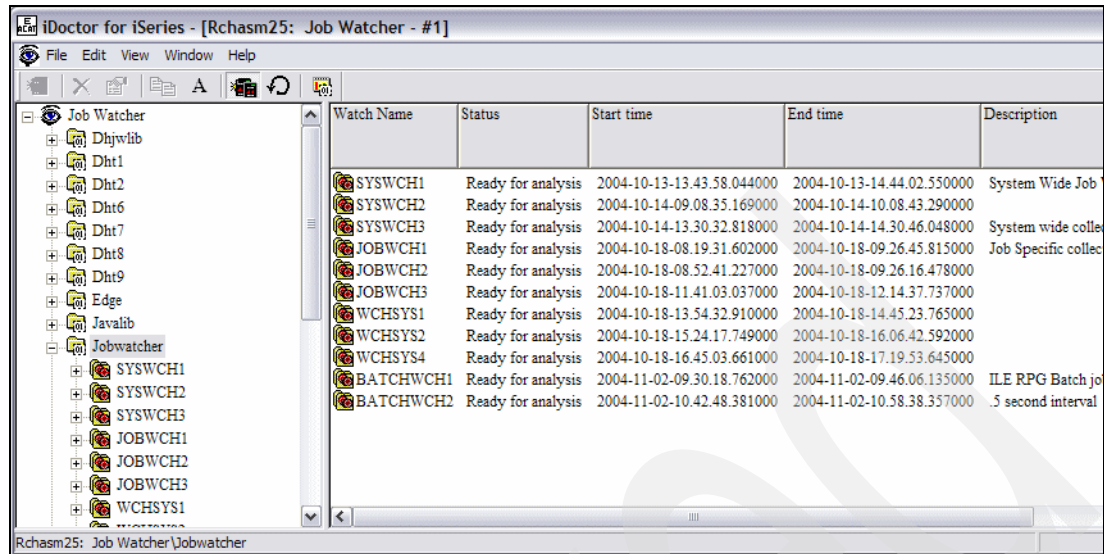


Figure 3-35 Job Watcher collection view

Figure 3-36 shows an example of four levels within the tree:

- All libraries on the system that contain Job Watcher output data.

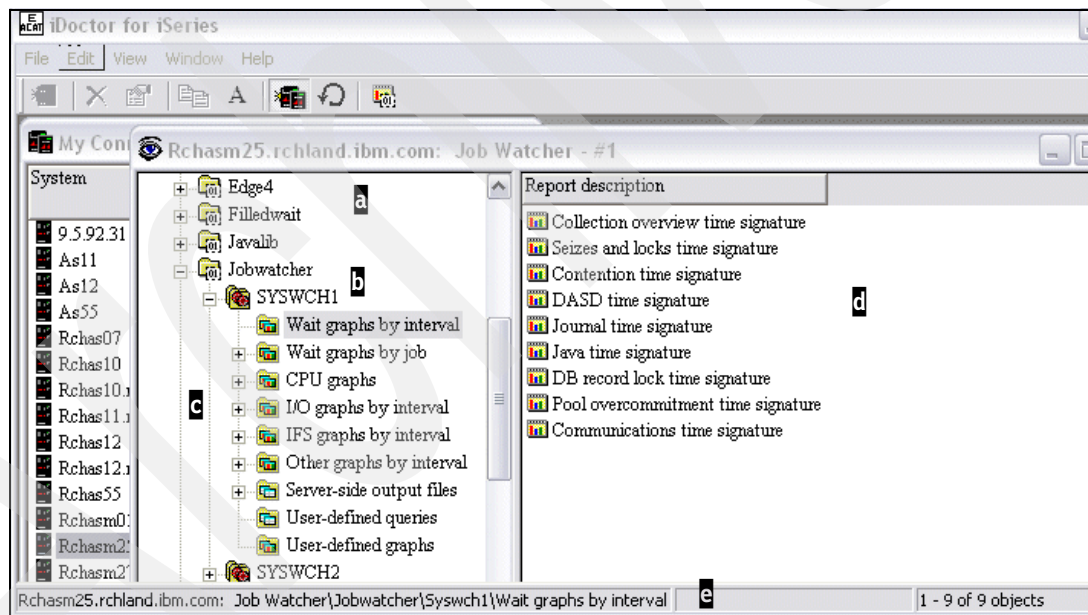


Figure 3-36 Job Watcher view layout

- The next level beneath the library is a list of Job Watcher collections. By right-clicking a Job Watcher collection, you can access the collection properties. In the property window, you can check the collection options that were used for this Job Watcher collection (Figure 3-37 on page 81 and Figure 3-38 on page 82).
- Below the collections is an initial list of folders: graphs, output files, and queries. By clicking on one of these initial folders, you see either another list of folders, a list of the server side output files, or user-defined queries and graphs.

- d. In our example, we clicked **Wait graphs by interval**, which produced the associated reports shown in the right pane.

Note that the Server side output files folder (left pane) lists the files that are part of or are associated with the collection. A user can open the file and see all records. Reports (in the right pane in Figure 3-36 on page 80) are either user-defined or Job Watcher-supplied table views over one or more files in Job Watcher. The order of the fields and selection of records are customized for the report's intent.

- e. At the bottom of the iDoctor for iSeries window is a view showing the level we have currently drilled down to.

Attention: Newly created libraries will not automatically appear under Job Watcher. To refresh your list of libraries, the easiest thing to do is to select (click) the Job Watcher folder within the tree in the left pane. This action refreshes the list automatically.

If the Job Watcher folder was already selected, clicking the refresh button is necessary to update the list of libraries.

Right-clicking a Job Watcher collection gives a fast path to the graph folders, plus additional options to Create idle thread graph files, Copy, Delete, and Transfer a collection.

Creating Idle threads graph files is discussed in “Waits that span multiple intervals” on page 267.

Copy, Delete, and Transfer options are discussed in Appendix D, “Overview of Job Watcher 5250 commands” on page 241. We discuss the CPYJWCOL, DLTJWCOL, and FTPJWCOL commands.

When right-clicking an active collection, there is also the option to stop the collection.

In Figure 3-37, we selected the **Properties** option.

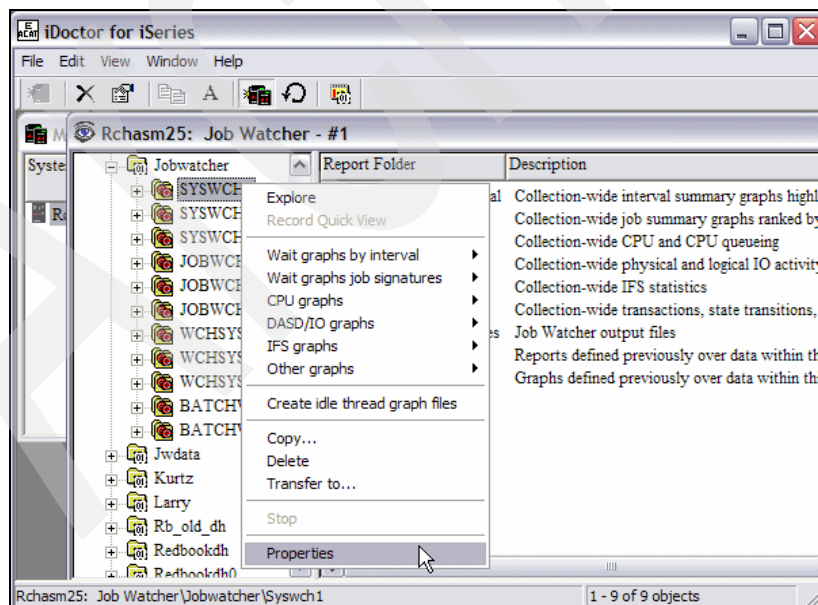


Figure 3-37 Job Watcher collection properties

In the Properties window, we can see the data options settings that were selected during the starting of the Job Watcher collection.

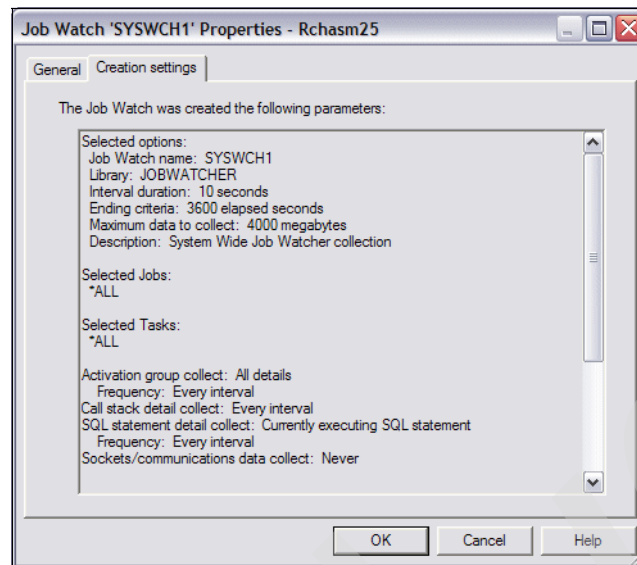


Figure 3-38 Collection settings

3.5.2 iDoctor preferences

Before we start reviewing some of the various graphs and reports within Job Watcher, we first take a look at the preference settings that can be used within the iDoctor client. From all of the windows that you enter via the iDoctor client, you can select **Edit** → **Preferences** to display the iDoctor preferences.

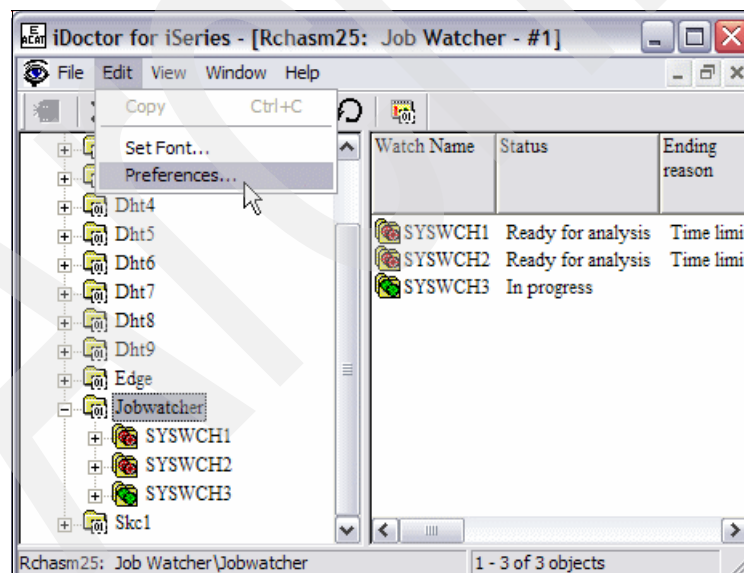


Figure 3-39 iDoctor client: Preferences

We do not cover all of the settings because they are all well documented within the iDoctor documentation. We, however, do cover those that we feel enhance your ability to analyze Job Watcher data.

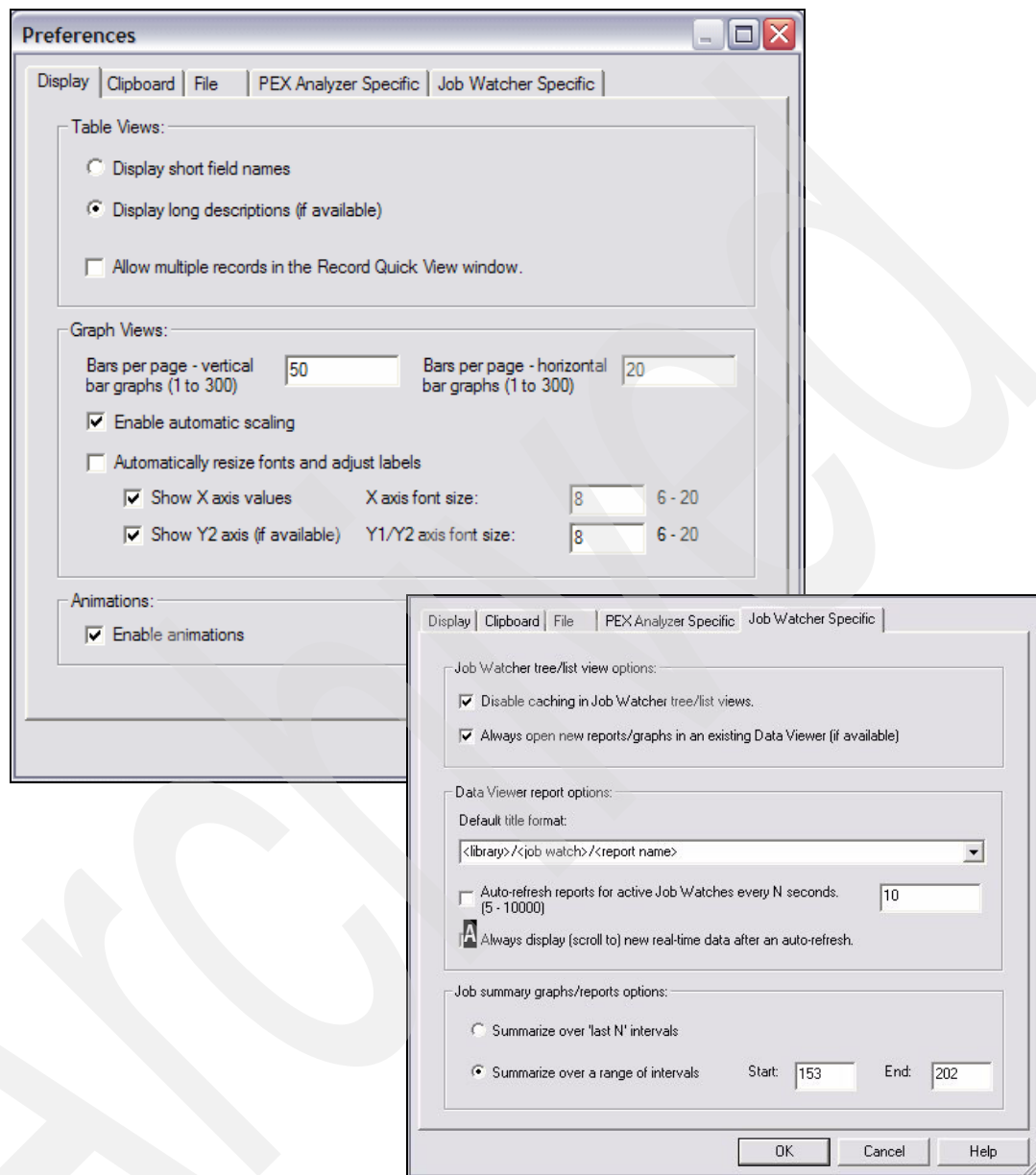


Figure 3-40 Display preferences: accessing example 1

You can also access the iDoctor preferences within a Job Watcher graph or report by right-clicking anywhere in the graph or report.

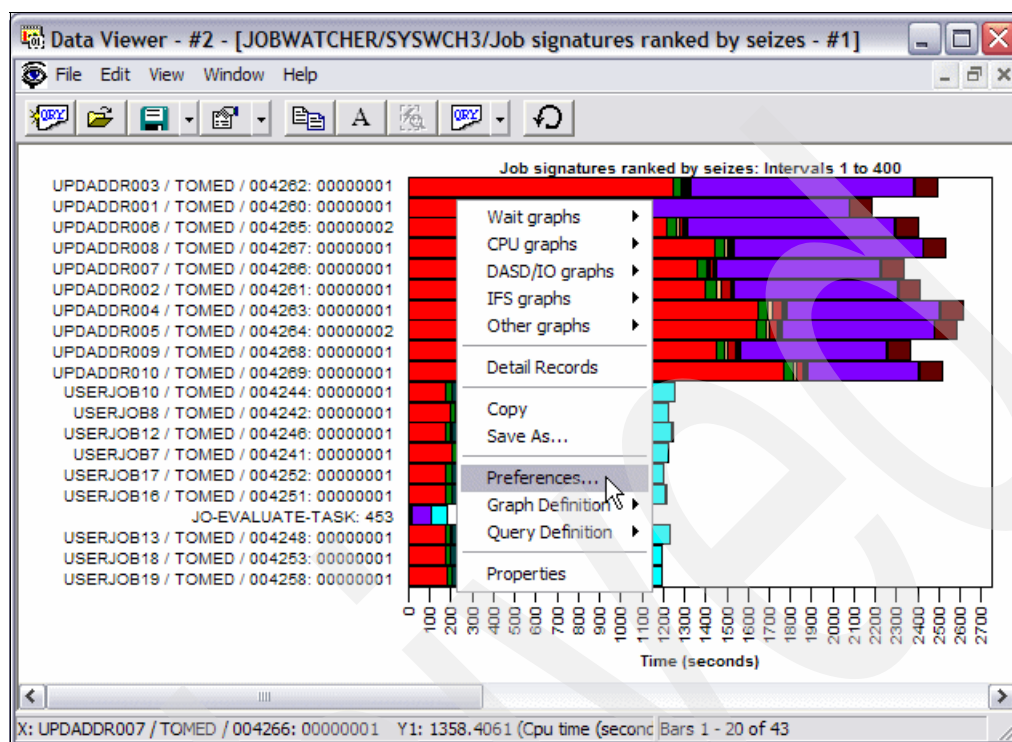



Figure 3-41 Display preferences: accessing example 2

- Bars per page (vertical and horizontal): Sometimes you will want to increase or decrease the number of bars that you display within your graph. This enables you to see more or fewer intervals. Separate values may be specified, one for horizontal-type bar graphs and one for vertical-type bar graphs.
- Enable automatic scaling: When turned on, all of the wait graphs by interval will have their y axis automatically adjust to the current highest value. When analyzing these graphs we have found that sometimes it is better to turn off this feature and manually adjust the maximum y axis value. The reason is that it is easy to scroll through a graph and not notice the y axis value increasing and decreasing and miss a problem.
- Automatically resize fonts and labels: In order to see all of the y axis legend descriptions on some of the larger graphs it is necessary to turn off this feature and make the y axis font smaller. Figure 3-40 on page 83 shows an example of where we unchecked Automatically resize fonts and changed the y axis font to 8 in order to see all of the legend descriptions.
- Job Watcher specific:
 - Auto-refresh reports: When viewing Job Watcher data in real time, make sure this value is turned on. Also, set the value to equal or more than your collection interval time. If your collection interval is 20 seconds, do not set the refresh time to 10 seconds.
 - Always display  (in Figure 3-40 on page 83): Again, when looking at real-time data, make sure this is set on as well.
- Job summary graphs/reports options:
 - Summarize over last N intervals: We found that this option is best used when analyzing real-time data.

- Summarize over a range of intervals: When analyzing a collection that is completed this option worked best for us. The range we use depends on the problem we were analyzing. For example, if the performance problem was present during the entire collection we would set the range from 1 to n where n would be the last interval in our collection. But if the performance problem we are trying to analyze was present during only a portion of the collection, we would then change the range to equal only the time the problem was present.

3.5.3 Reports and graphs: Example 1

How you start viewing your Job Watcher data depends on the type of problem you are faced with, and sometimes we are faced with a totally unknown entity. This is where Job Watcher really excels in pointing us to the source of a problem.

Important: Job Watcher will not always identify the exact cause of a problem, but if you think of the problem as a large building, Job Watcher at least leads you to the right office.

Job Watcher comes with an extensive collection of graphs. In this section we introduce you to some of the graphs used in analyzing Job Watcher data originally collected for all jobs and system tasks.

1. Figure 3-42 shows a list of the different graph folders in the left window pane; in the right window pane is a list of graphs currently listed under Wait graphs by interval folder.

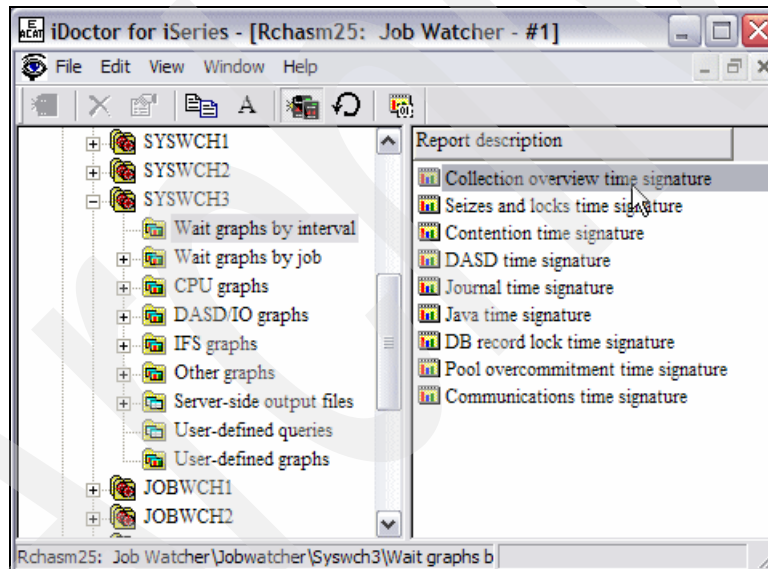


Figure 3-42 Job Watcher graph folders

2. For the purpose of this exercise, we assume that we have very little information about the problem. All we know is that there was a performance problem and a Job Watcher collection was collected during this time.

The first graph that we want to start with a case like this is titled *Collection overview time signature*, which will show the run and wait times for all active jobs during each interval. Some of what we will see in this graph can also be found by using other iSeries commands such as WRKACTJOB, WRKSYSSTS, WRKDSKSTS, WRKSYSACT, as well as Performance tools reports. But we demonstrate in the following examples that Job Watcher enables us to drill down through a suspected wait problem to determine the impact that this wait condition is having on the entire system or on a particular job.

Figure 3-43 shows an example of a Collection overview time signature graph. We use this figure to illustrate areas of information about the typical multi-colored graph and some ways to manipulate the graph data shown. Note, however, that not all Job Watcher graphs will have the same options.

We call your attention to the areas discussed below. You can customize the way some of the information is shown, but the colors representing each specific metric are fixed per metric. The best way to understand these descriptions (and the real Job Watcher graphics) is to view this book in a PDF viewer supporting colors to see the unique colors used for each metric. These colors are less distinguishable in black-and-white.

- The left y-axis represents clock time seconds. The range of time values can be increased and decreased.
- The right y-axis represents the number of active threads or tasks for the jobs selected to be included in the collection. The black line graph shows the number of active threads or tasks over the set of time intervals shown.
- Select **Edit** → **Preferences** to specify how many intervals (bars) are shown on a single graph window and the graph text font size to minimize the amount of scrolling left-right and up-down that is required to see the information being displayed.
- In this example, each bar represents the total threads or tasks wait times within each interval for all jobs, tasks, or threads. Job Watcher has specific colors for each wait type, as shown in area **1** in the graph.
- The interval time stamp for a subset of interval bars is listed vertically under that bar.
- For our Collection overview time signature example, the matrix shown as taking the highest amount of time is purple - Seize contention time (seconds) (as indicated by portions of the interval bars represented by the **A** areas) and red CPU time (seconds) (as indicated by **B** areas).
- You can scroll through all time intervals using the slider at the bottom of the window.

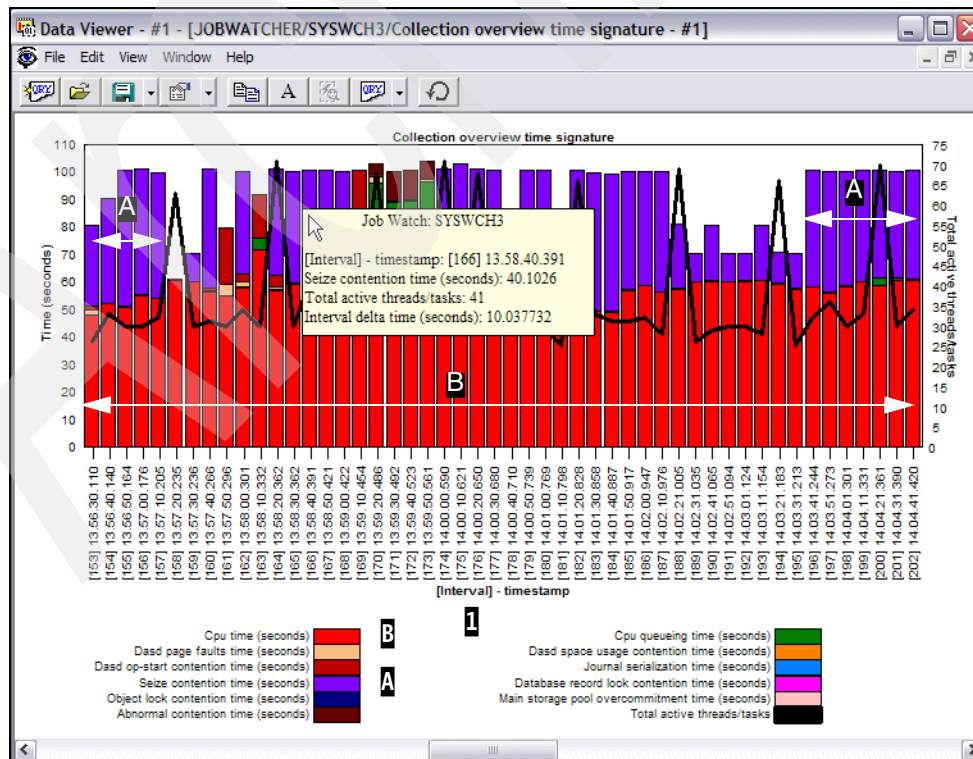


Figure 3-43 Collection overview time signature

The help window shown in this example displays whenever you move your cursor over one of the sections within a bar of the graph. The purpose of this window is to display extended information about the interval and the wait condition your mouse is positioned over.

Important: The height of the bars for each interval may vary. By design, Job Watcher has a default set of wait buckets to be included (and shown) for each of its graphs, such as Collection overview time signature shown here with the 12 wait bucket metrics. For a particular collection, since not all wait buckets are shown for an interval, the heights of the bars may vary. This is also shown in our example.

The Run/wait time signature for a single job includes all 32 wait buckets and the interval bars are all the same height, as shown in Figure 3-46 on page 89.

- When you see an interval with a high wait condition or high CPU usage, you will probably want more details about which jobs are being effected. Right-click the next level of graphs to display. Figure 3-44 shows an example.

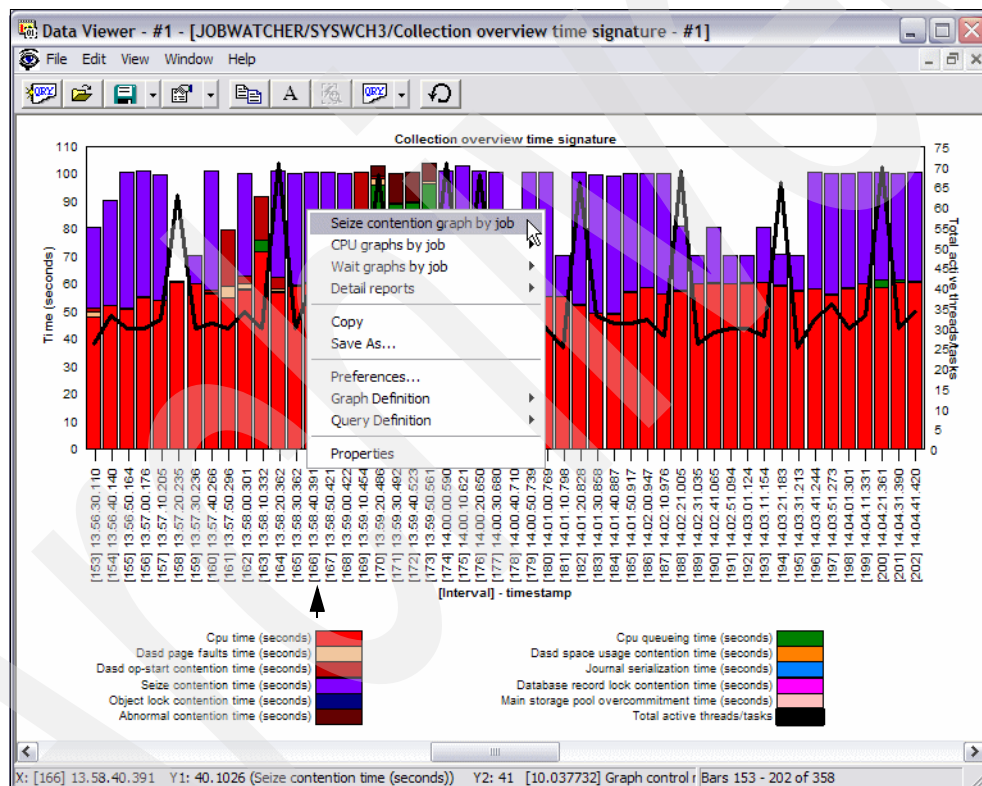


Figure 3-44 Additional graph options

In Figure 3-44, when we right-clicked the Seize contention section of the graph, the first graph option displayed is a Seize contention graph by job. The first available graph always applies to the section you clicked on.

4. An example of a seize contention graph by job is shown in Figure 3-45. In this graph we show all of the jobs that were waiting on a seize during interval 166.

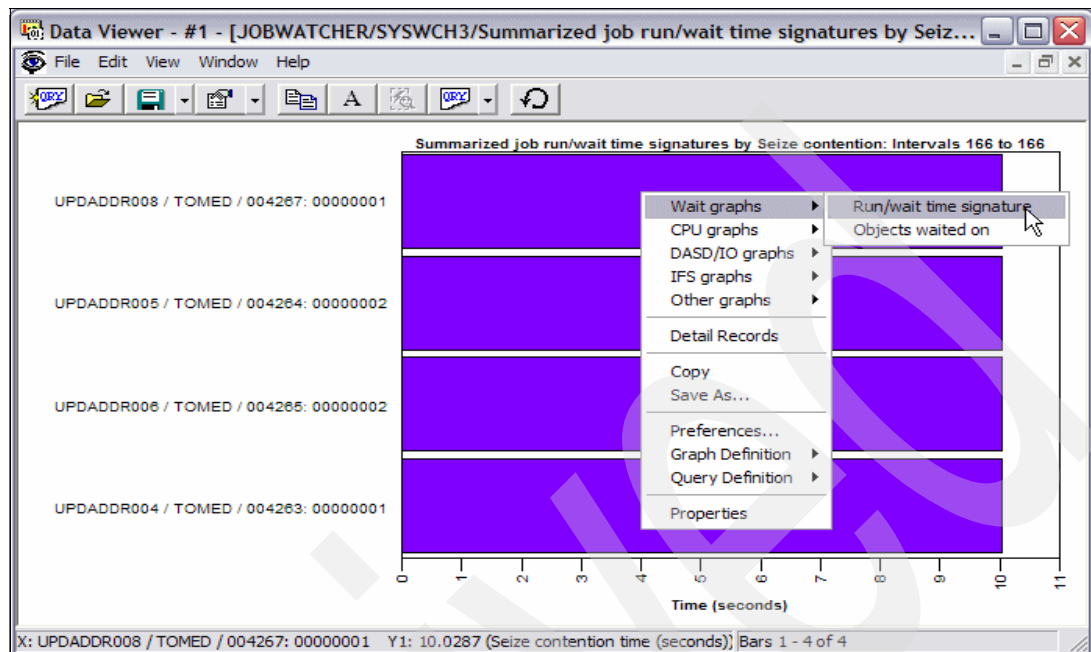


Figure 3-45 Summarized jobs by seize contention

The Summarized job run/wait time signatures by Seize contention lists all jobs waiting on a seize in descending order by the wait time. In the example in Figure 3-45, this is not obvious as all four jobs are waiting about 10 seconds, which was also the length of our collection interval. Right-click on any one of these jobs to display additional graph options. You can also display the Detail Records. Selecting the Run/Wait time signature graph displays a complete run/wait signature for the selected job. In this example we have selected the first job listed.

- When you display a job run/wait signature, the graph always starts at interval one. Scroll to the right until you reach the interval you want to investigate. In our example we are interested in looking at interval 166 (Figure 3-46).

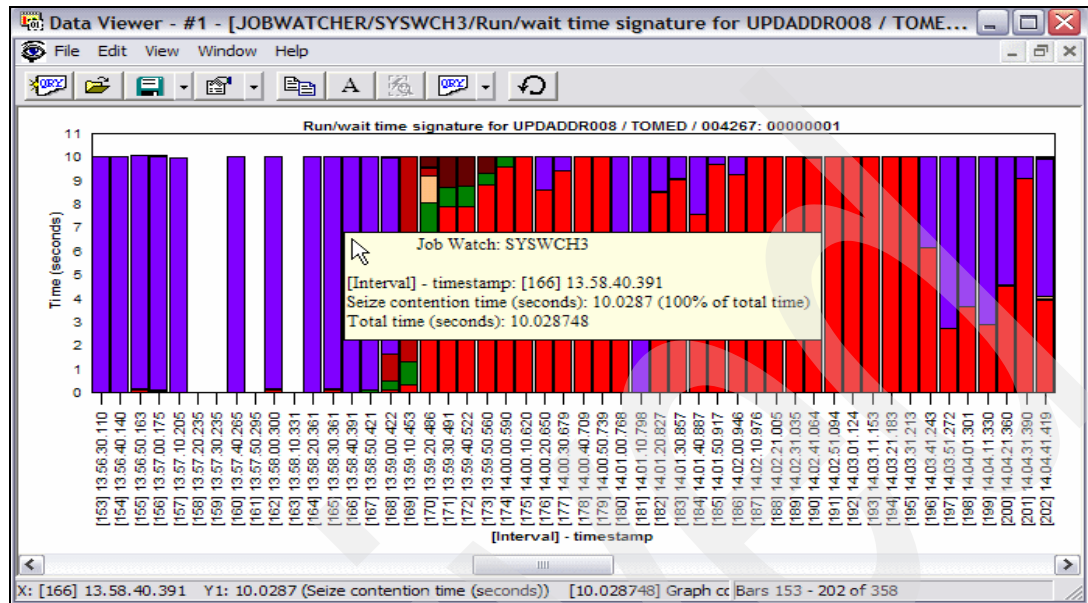


Figure 3-46 Job run/wait signature

Important: In Figure 3-46 there are four intervals—158, 159, 161, and 163—where no graph bars are displayed. This is because job task count details are collected only when a job uses CPU. If a job is in a long wait across multiple intervals you will see blank time intervals between solid waits. For the means to correctly populate these blank intervals with bad waits that are occurring, see “Waits that span multiple intervals” on page 267.

By positioning our cursor over interval 166 we can see that this job spent 100% of this interval sample waiting on a seize.

- Click interval **166** to display the Interval Details window, which has 12 tabs that summarize the data collected for this specific job at this specific interval. Read more about the various tabs in 3.5.5, “Tabs in the Interval Details window” on page 97.

The first tab displayed in the Interval Details window is the Quick view. The holding thread is displayed in the traditional OS/400 format of job name/user/job number and the *task count* is in parenthesis at the end.

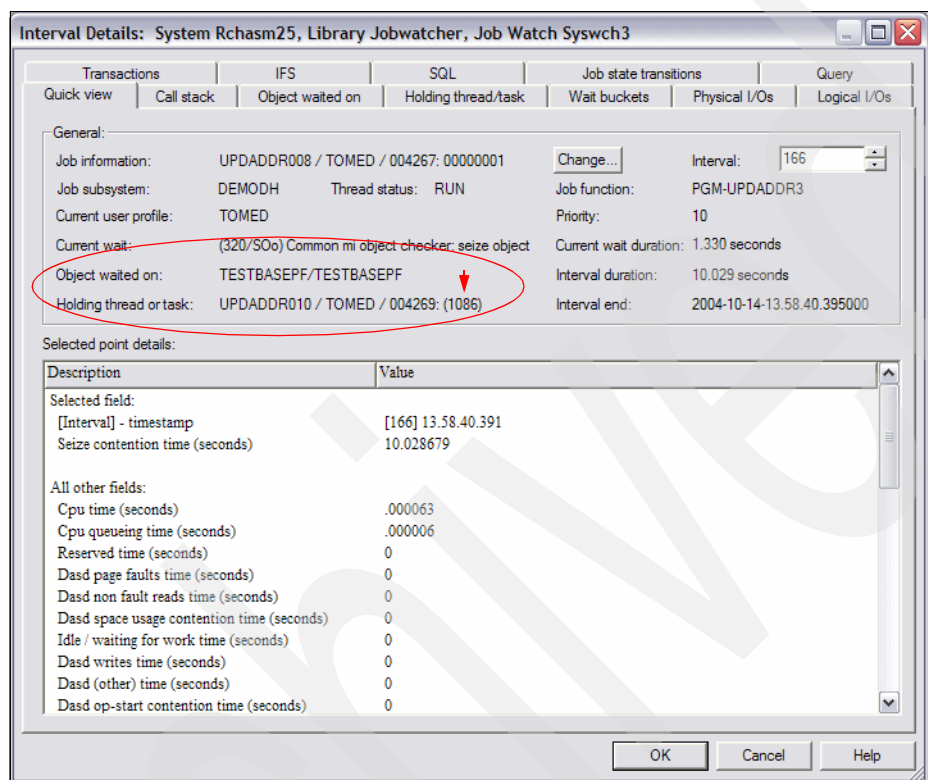


Figure 3-47 Job/interval details: task count 1084

7. Now that we have the holding thread or task, the next step is to check it to determine the status of this job or task in interval 166. There are two ways to do this:
 - a. Query the Job Watcher output file QAPYJWTDE. We discuss creating custom queries and some that we have created in Appendix C, “Querying and graphing tips for Job Watcher” on page 221.
 - b. Use the **Change** button on the Interval Detail window to change the job or task that is displayed. When you click the Change button, a window opens for selecting the job you want to display, as shown in Figure 3-48.

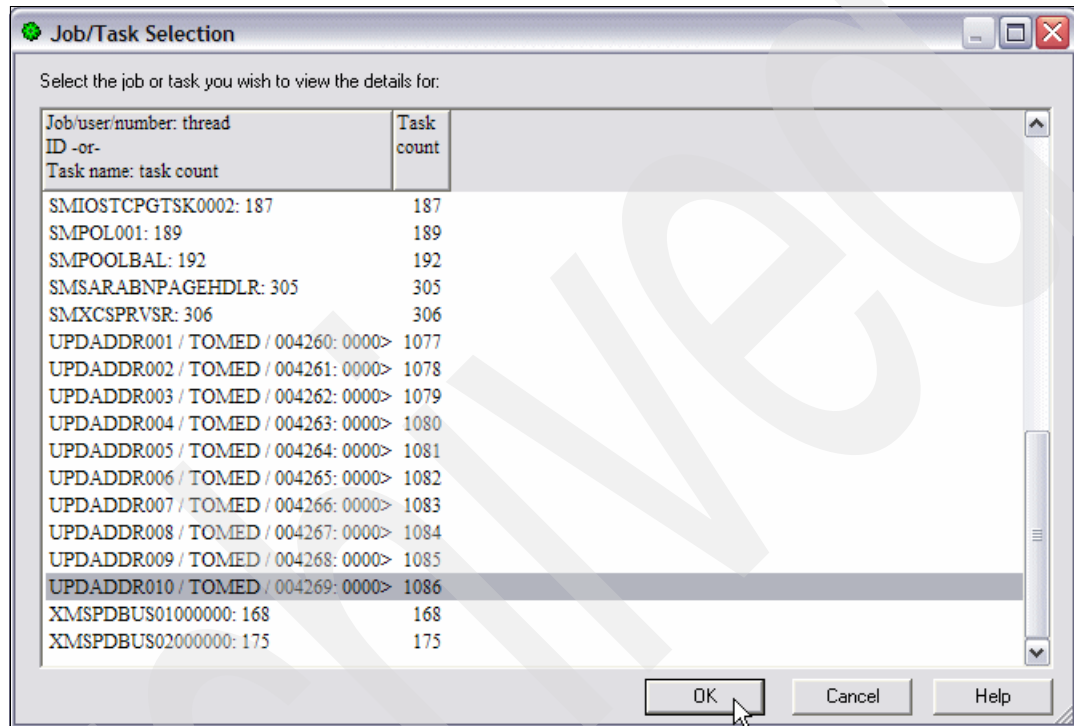


Figure 3-48 Select job or task to view

Note: The time it takes to display the contents of this window is proportional to the number of active jobs, threads, or tasks you specified to include in the Job Watcher collection.

8. Click **OK** to see the job or task details shown in Figure 3-49 on page 92.

9. Figure 3-49 shows the Interval details for task count 1086. At a quick glance it would be easy to think that this task count is also waiting, as the current wait is shown as QGa Qugate^(A).

In reality, however, this field could be either the current or the task's most recent (last) wait. In the case of task count 1086, it is the last wait. You can tell it is the last wait by observing the Thread status as RUN ^(B) and the Current wait duration is 0 microseconds ^(C).

Note that the thread number (00000001, in our example) comes from the THREADID field in the primary Job Watcher database file QAPYJWTD. (File APYJWTDW is described in "The master file QAPYJWTD for jobs, threads, and tasks" on page 208.)

Interval Details: System Rchasm25, Library Jobwatcher, Job Watch Syswch3

Transactions	IFS	SQL	Job state transitions	Query
Quick view	Call stack	Object waited on	Holding thread/task	Wait buckets
				Physical I/Os
				Logical I/Os

General:

Job information: UPDADDR010 / TOMED / 004269: 00000001 Change... Interval: 166

Job subsystem: DEMODH Thread status: RUN ^(B) Job function: PGM-UPDADDR3

Current user profile: TOMED Priority: 10

Current wait: ^(A) 2/QGa) Qugate, not otherwise identified Current wait duration: 0 microseconds ^(C)

Object waited on: Segment type MACHINE INDEX RADIX4 SECONDAR Interval duration: 10.029 seconds

Holding thread or task: UPDADDR005 / TOMED / 004264: (1081) Interval end: 2004-10-14-13.58.40.392000

Selected point details:

Description	Value
Selected field:	
[Interval] - timestamp	[166] 13.58.40.391
Seize contention time (seconds)	0
All other fields:	
Cpu time (seconds)	10.029124
Cpu queueing time (seconds)	0
Reserved time (seconds)	0
Dasd page faults time (seconds)	0
Dasd non fault reads time (seconds)	0
Dasd space usage contention time (seconds)	0
Idle / waiting for work time (seconds)	0
Dasd writes time (seconds)	0
Dasd (other) time (seconds)	0
Dasd op-start contention time (seconds)	0

OK Cancel Help

Figure 3-49 Interval 166 details: task count 1086

10. Also, by clicking the **Wait buckets** tab we can see that task count 1086 spent 100% of the interval consuming CPU.

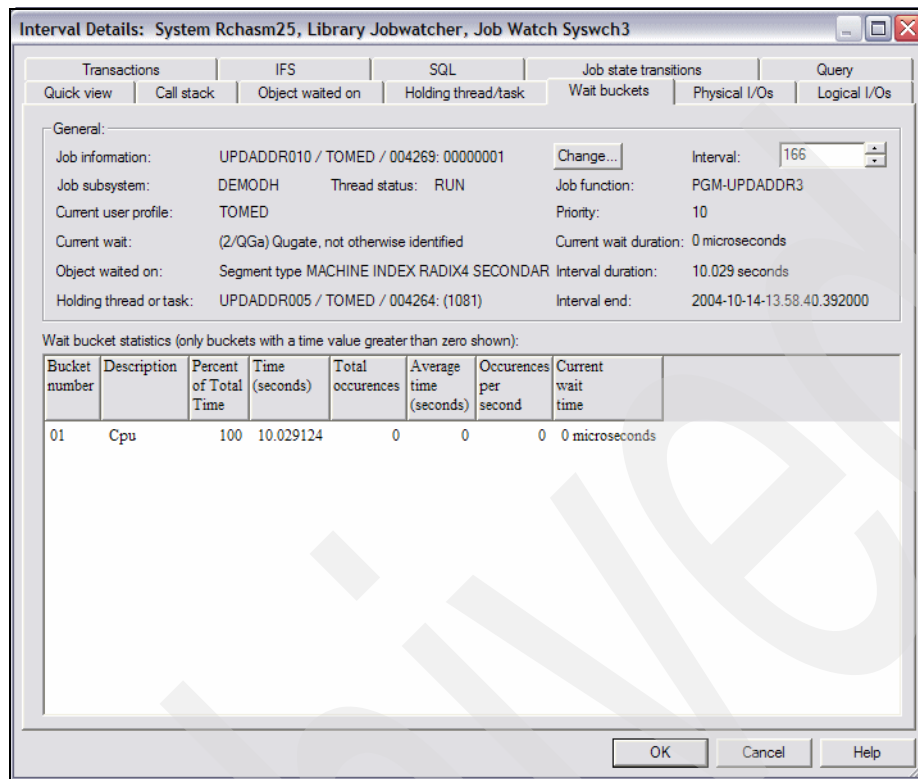


Figure 3-50 Wait Buckets for task count 1086, interval 166

11. Click the **Call stack** tab to see an example of the call stack at the end of this sample interval, as shown in Figure 3-51.

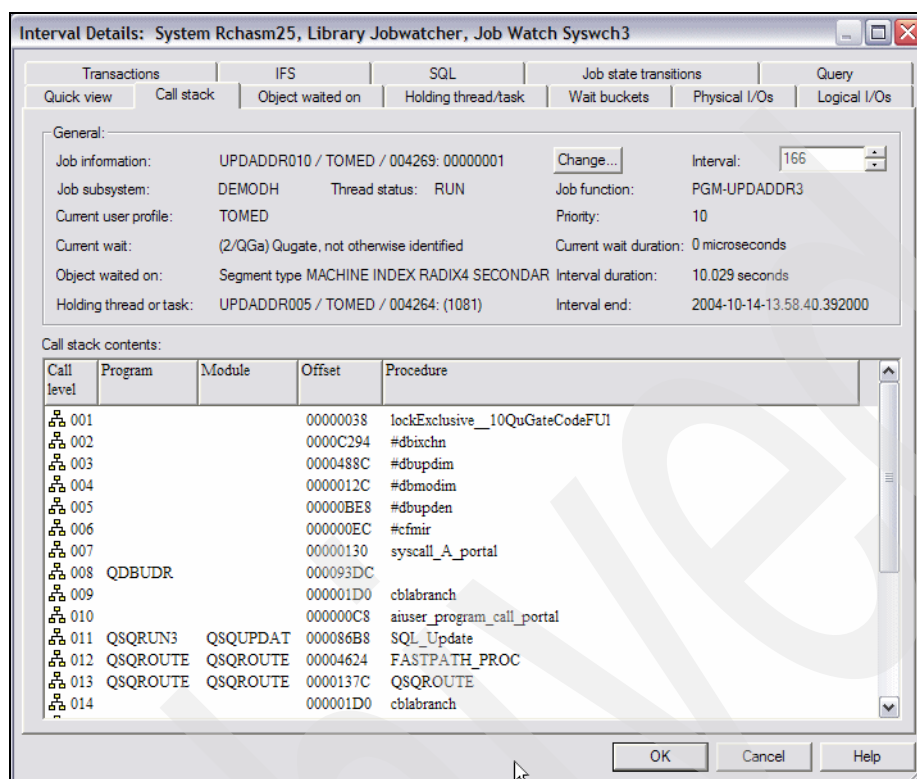


Figure 3-51 Call stack for task count 1086, interval 166

Given the call stack information, even if you are not familiar with the application, you can show this information to those who are familiar with the application implementation. They should be able to tell you what the job is doing, which helps you understand whether what you see is normal.

In our example, looking only at the program names that are listed, we can tell that a DB2 UDB for iSeries (OS/400 database) function is being used (QDBUDR) along with SQL functions as indicated by the QSQ prefix on the program names.

3.5.4 Graphs and reports: Example 2

In the first example of reports and graphs, we demonstrated how to use a drill-down methodology to determine the source of a problem. In this example, we show some ways to minimize the multiple drill-down steps.

Using the UPADDR010/TOMED/004269 job, we have shown that there is significant seize wait time that must be understood. In Figure 3-43 on page 86 we looked at a Collection summary graph, in which it is obvious that there is a significant amount of seize time.

In this example we show another way to use specific graphs to determine the source of a problem. In Figure 3-52 we select the **Wait graphs by job** folder and the **Job signatures ranked by seize** graph, which leads to the example window in Figure 3-53.

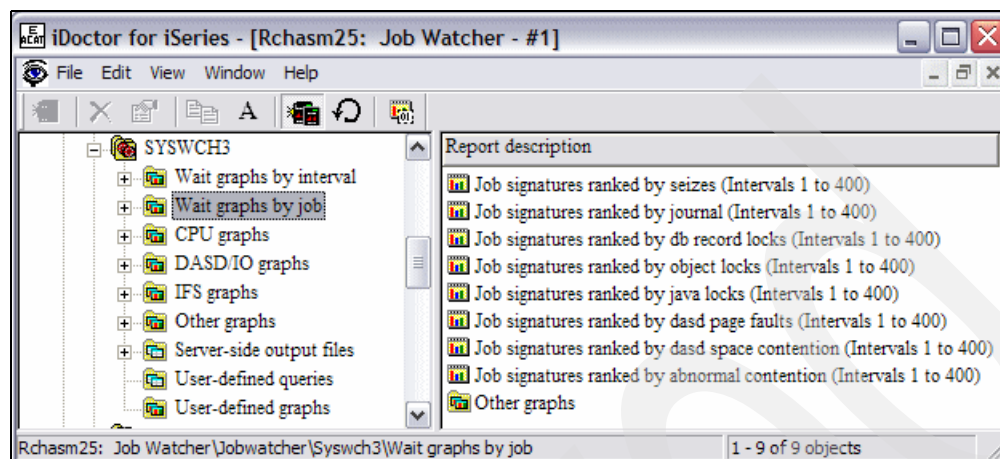


Figure 3-52 Wait graphs by job

Figure 3-53 shows all of the jobs in the Job Watcher collection listed in descending order of seize time.

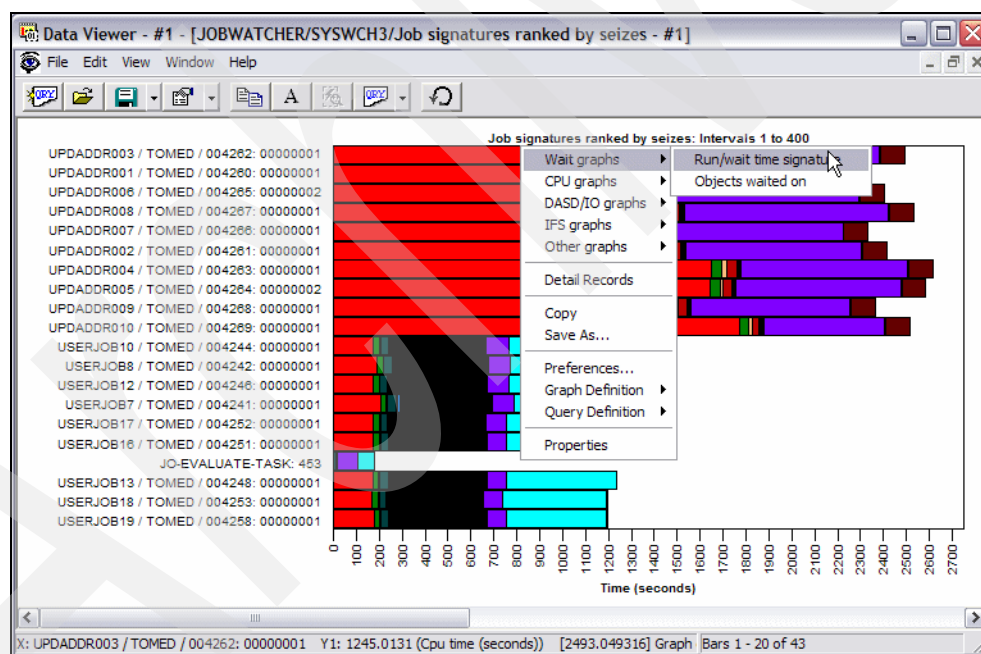


Figure 3-53 Job signature: seizures

Right-click on the first job and select **Wait graphs** → **Run/Wait signature** to display the job's complete run/wait signature for the entire Job Watcher collection (Figure 3-54 on page 96).

Remember, in most of Job Watcher's run/wait signature windows, purple indicates seizes (A).

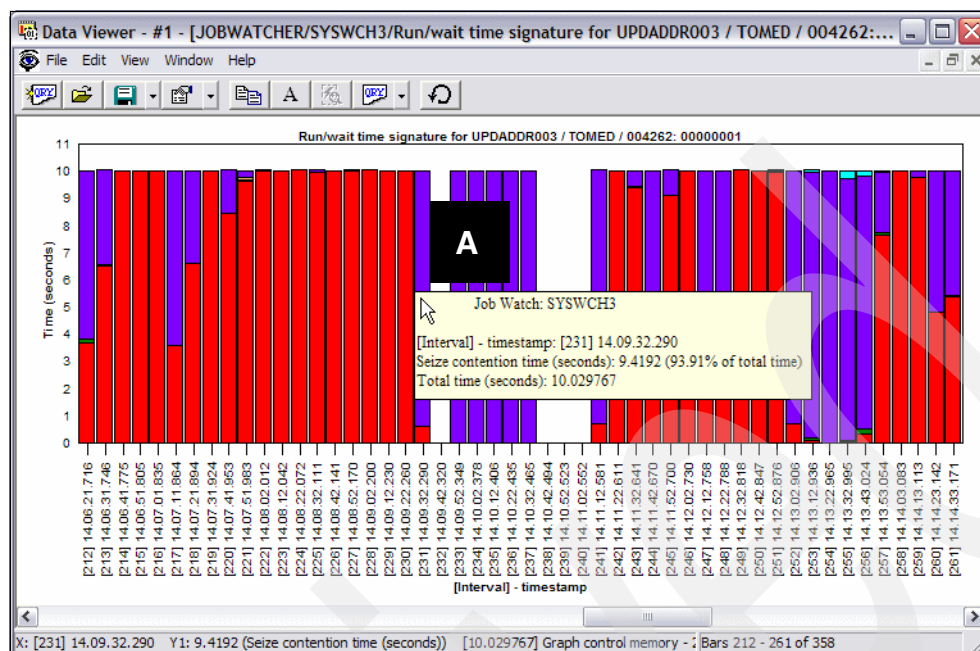


Figure 3-54 Run/Wait time signature for job UPDADDR003

Select an interval with a high amount of seize time by clicking on the interval. In Figure 3-54 we selected interval 231 to display the Interval details window.

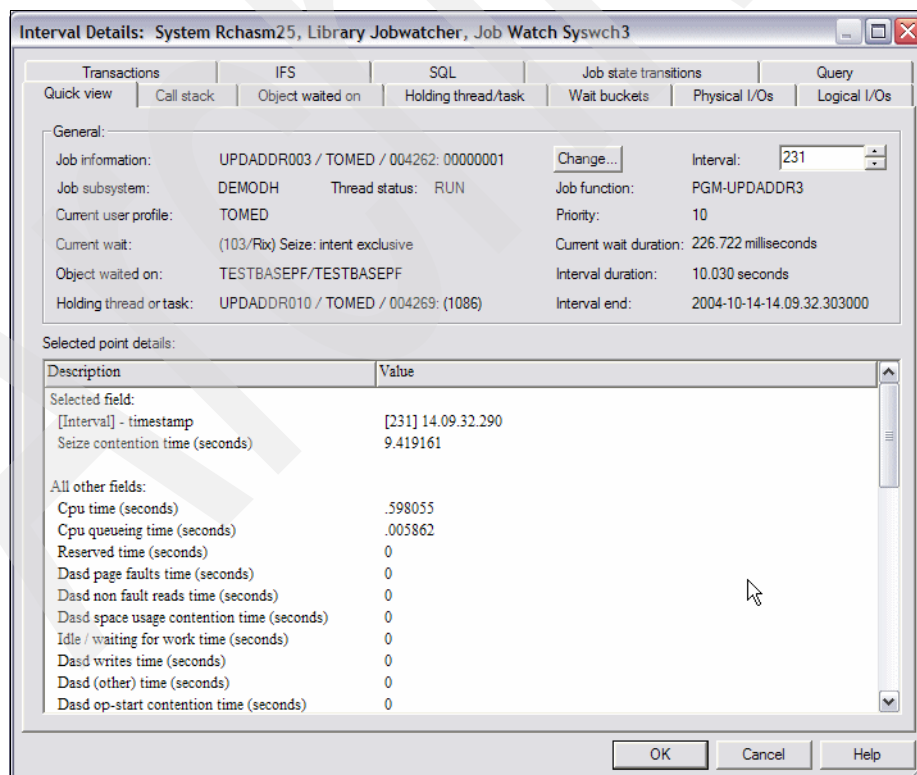


Figure 3-55 Interval details

In this example the holder is also task count 1086, as in the first example.

3.5.5 Tabs in the Interval Details window

In this section we explain the tabs in the Interval Details window.

The General pane (a) appears in the top part of all panels and displays the current job information and interval. Use the **Change** button for different job information, as described in step b on page 91.

► Quick view tab

Selected point details (b): This is unique to the Quick view panel and is called select point details because the first section (Selected field text) varies depending on the location within the graph you select. The second section (All other fields text) is a list of all other wait fields displayed in seconds. Scrolling down the Quick view panel is the interval start and end time of day. Also displayed is the job or task name, task count, and the total elapsed time for the interval.

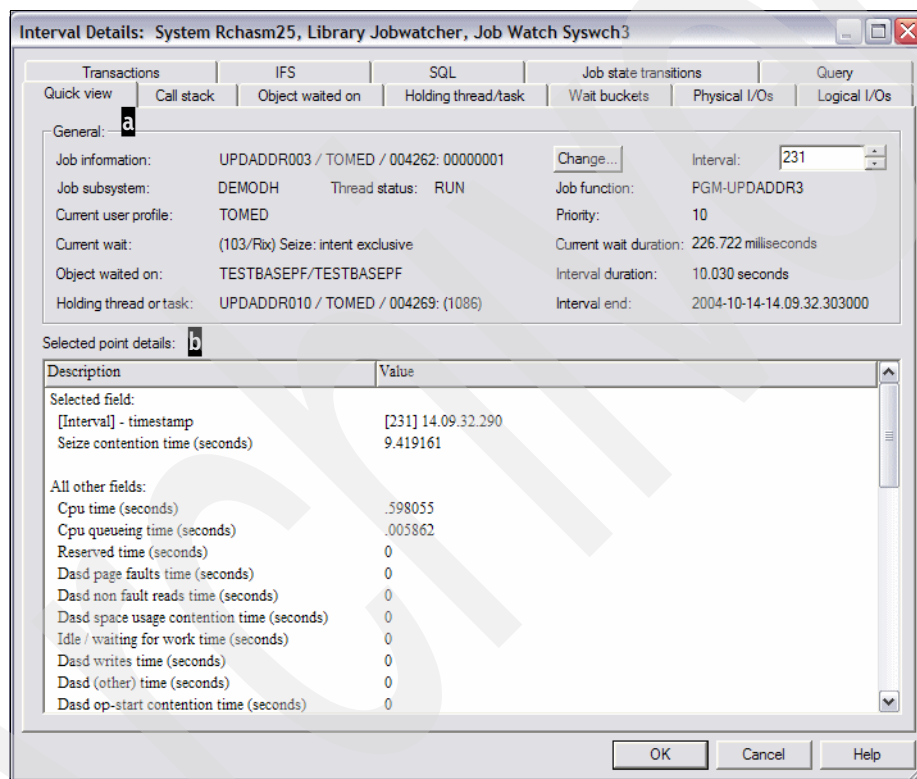


Figure 3-56 Interval Details: Quick view tab

► Transactions

Transaction detail appears only for jobs doing 5250 display I/O. This kind of a job is often referred to as an *interactive workload* or *interactive job*. When analyzing an interactive response time problem, this panel is useful in determining the effect the problem had on this user.

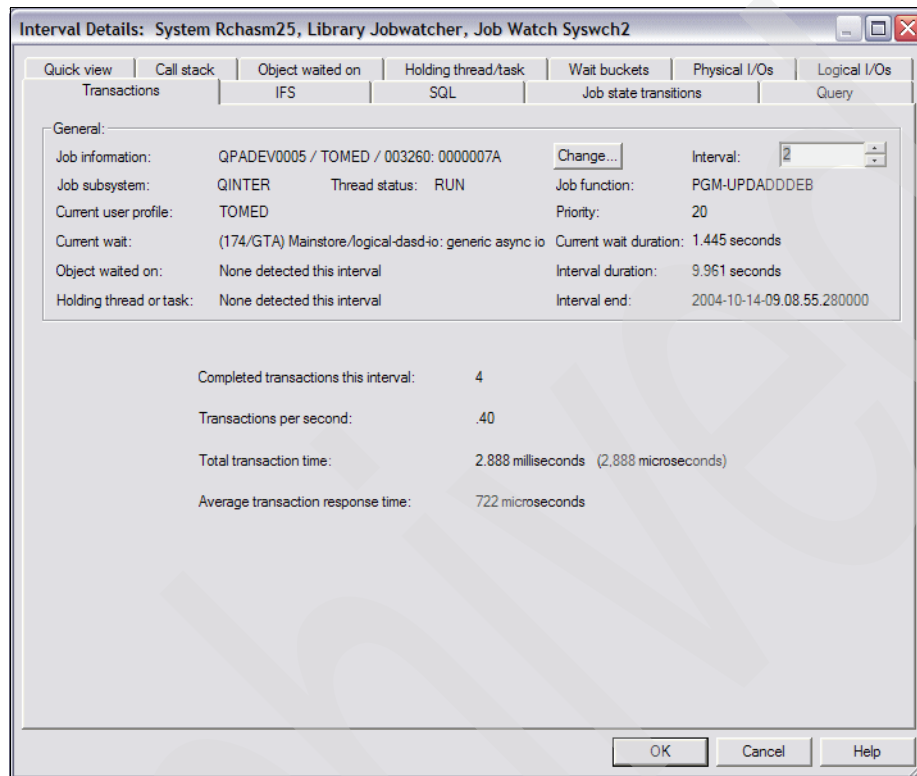


Figure 3-57 Interval Details: Transactions tab

► Call stack

When collecting call stack details, we have to remind ourselves that Job Watcher is not a trace tool. It is a sampling tool, which is what makes Job Watcher *nonintrusive*. This means that Job Watcher will not interrupt a job in order to harvest any job information including call stack details. As a result we may not always be able to harvest the call stack on a multiprocessor system.

The call stack in Job Watcher is displayed in order with the newest program or module being listed first. This is the opposite of what you may be used to seeing when using the DSPJOB command.

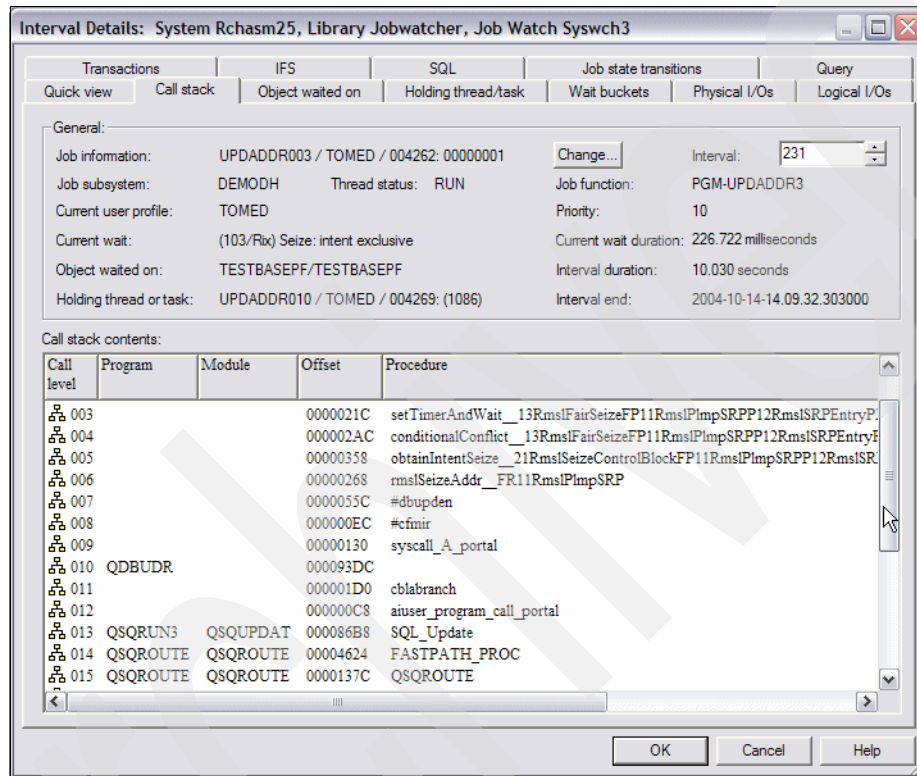


Figure 3-58 Interval Details: Call stack tab

► IFS tab

The IFS panel displays I/O counts and average times that occurred during this interval snapshot against objects that resided on the IFS.

Interval Details: System Rchasm25, Library Jobwatcher, Job Watch Syswch3

Quick view	Call stack	Object waited on	Holding thread/task	Wait buckets	Physical I/Os	Logical I/Os
Transactions	IFS	SQL	Job state transitions	Query		

General:

Job information: UPDADDR003 / TOMED / 004262: 00000001 [Change...](#) Interval: 231

Job subsystem: DEMODH Thread status: RUN Job function: PGM-UPDADDR3

Current user profile: TOMED Priority: 10

Current wait: (103/Rix) Seize: intent exclusive Current wait duration: 226.722 milliseconds

Object waited on: TESTBASEPF/TESTBASEPF Interval duration: 10.030 seconds

Holding thread or task: UPDADDR010 / TOMED / 004269: (1086) Interval end: 2004-10-14-14.09.32.303000

IFS Reads:			IFS Opens:			IFS Creates/Deletes:		
	Count	IOs/second		Count	IOs/second		Count	IOs/second
Symbolic link reads:	0	0	Opens:	0	0	Directory creates:	0	0
Directory reads:	0	0				Non-directory creates:	0	0
						Directory deletes:	0	0
						Non-directory deletes:	0	0

IFS Lookup cache:		
	Count	IOs/second
Hits:	0	0
Misses:	0	0

OK Cancel Help

Figure 3-59 Interval Details: IFS tab

► Object waited on

The Object Waited on tab shows the type of object the job is waiting on. Figure 3-60 shows an object named TESTBASEPF that is a physical file.

In this case the wait object name is displayed twice because it is a physical file whose member name is the same as the file name. This is not always the case, especially for multi-member files or single member files whose member name is not the same as the file name.

Transactions		IFS		SQL		Job state transitions		Query	
Quick view	Call stack	Object waited on	Holding thread/task	Wait buckets	Physical I/Os	Logical I/Os			
General:									
Job information:	UPDADDR003 / TOMED / 004262: 00000001					Change...	Interval:	231	
Job subsystem:	DEMODH	Thread status:	RUN		Job function:	PGM-UPDADDR3			
Current user profile:	TOMED					Priority:	10		
Current wait:	(103/Rix) Seize: intent exclusive					Current wait duration:	226.722 milliseconds		
Object waited on:	TESTBASEPF/TESTBASEPF					Interval duration:	10.030 seconds		
Holding thread or task:	UPDADDR010 / TOMED / 004269: (1086)					Interval end:	2004-10-14-14.09.32.303000		
Wait object name:	TESTBASEPFTESTBASEPF								
Wait object type description:	PHYSICAL FILE MBR - DATA PART								
Wait object segment type description:	BASE MI SYSTEM OBJECT								
Advanced:									
Wait object type (hex):	0B90		Wait object segment type (hex):	0001					
LIC wait object:	Rix		LIC wait object handle:	27252918DD000000					

OK Cancel Help

Figure 3-60 Interval Details: Object waited on

► SQL tab

SQL details will be available only if you have selected to include SQL data as part of your data collection options. For additional information about including SQL data see 3.3.4, “Data collection options” on page 58.

SQL details include:

- SQL package name, location, and source date (shown at **A**).
- The actual SQL statement (shown at **B**). Depending on the data collection option selected when setting up the collection, this can be either the last or current SQL statement. If current was selected, then the SQL statement will display only as long as the SQL statement is active.

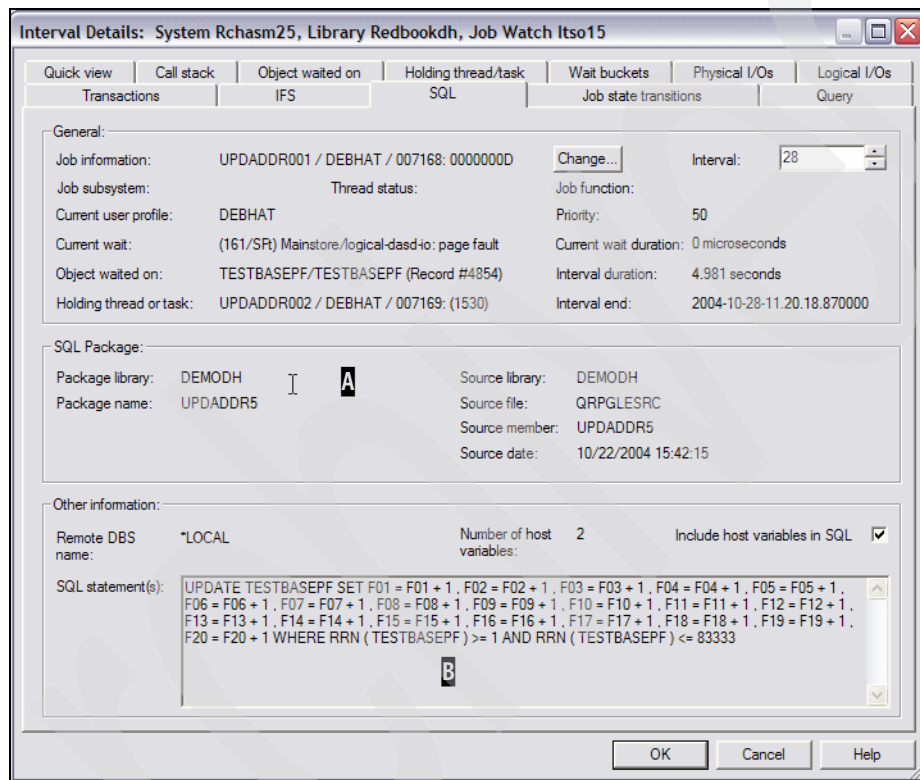


Figure 3-61 Interval Details: SQL tab

► Holding thread/task tab

The Holding thread/task tab currently gives only the thread task count, which is also available in the General section.

The Holding thread information is valid only if the Current wait duration is greater than zero (that is, the waiter was sampled while it was waiting), the Current wait is for a seize or lock, and Thread status is RUN or LCKW.

Note that gates, disk I/O, and other blocking codes have no holding thread number.

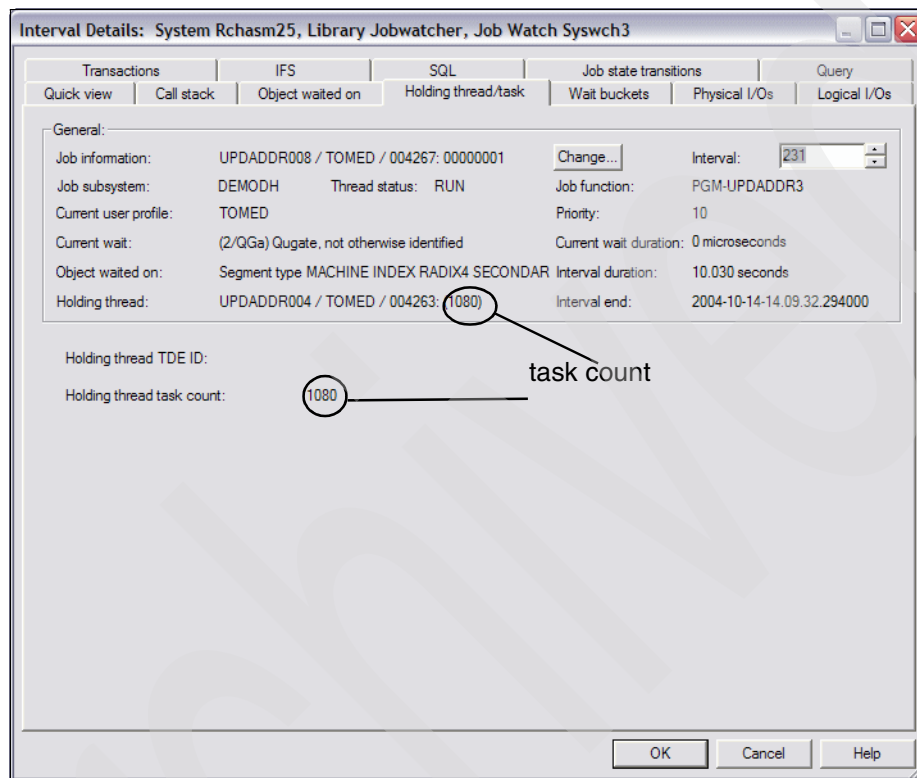


Figure 3-62 Interval Details: Holding thread/task tab

► Wait buckets tab

The Wait buckets tab lists all of the wait buckets for a thread or task where wait time was greater than zero:

- Bucket number - Currently there are 32 different wait buckets.
- Description - See 2.9, “LIC queuing primitives and more granular wait points” on page 32 for a detailed description of wait buckets.
- Percentage of Total Time - Applies to the interval being displayed.
- Time - The total time in seconds spent in a given bucket during the displayed interval.
- Total occurrences - Currently total occurrences are completed occurrences. A wait must be satisfied in order to register as an occurrence. The occurrences can be zero if the wait extends into the next interval. The time would still be accumulated and shown in the time column.
- Average time - Average time for each occurrence.
- Occurrences per second - Number of occurrences per second.
- Current wait time - If a thread was in a wait condition when the interval snapshot was taken, this is the cumulative time that the thread has been in the wait. The current wait duration has been included in the wait bucket total.

For more information, see the advanced topics in Appendix E, “Job Watcher advanced topics” on page 255.

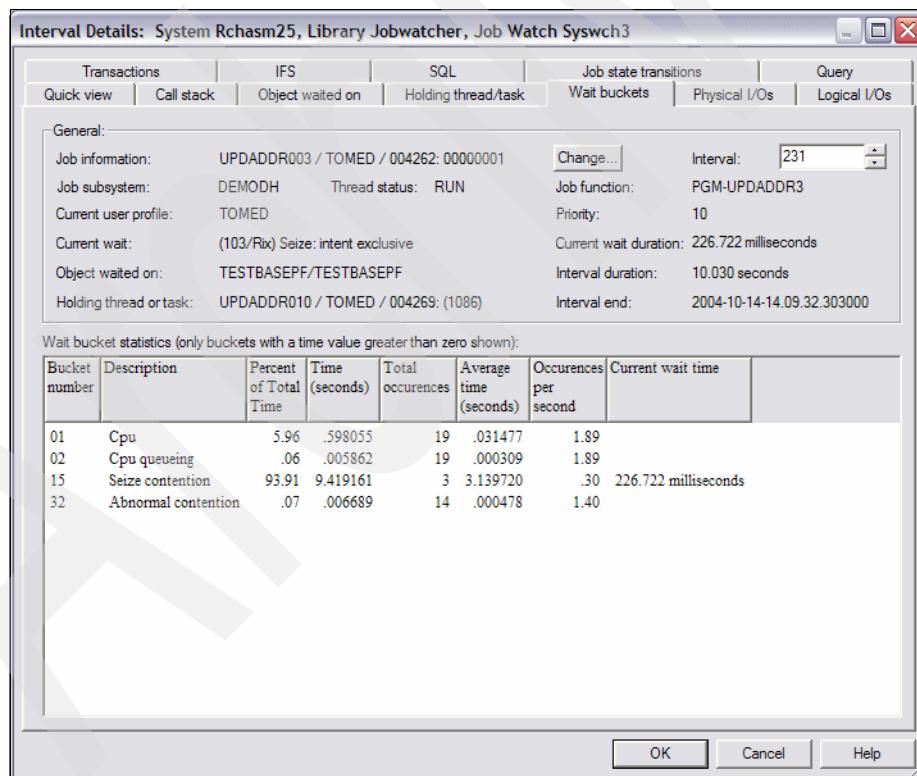


Figure 3-63 Interval Details: Wait buckets tab

► Job state transitions tab

In our example, interval 231 has 0 transitions. This is because, using the wait bucket statistics in Figure 3-63 on page 104, the only waits collected were seize contention waits (3) and abnormal contention waits (14). These kinds of waits do not cause the job, thread, or task to leave an activity level slot (no transition occurs).

A typical reason for the Active state to ineligible state transitions field to appear as greater than 0 is when the pool's activity level is set too low. Consider increasing the activity level.

However, in a very busy system, even with a reasonably high activity level value, you would still see a relatively low value under Transitions/second. This would be considered normal. At the risk of oversimplifying this activity level tuning knob, consider using the system value QPFRADJ set to either 2 (Adjustment at IPL and automatic adjustment (during normal activity)) or 3 (Automatic adjustment) when installing a new application workload. Observe the changes in activity level and pool sizes over a reasonable time and assess performance.

In a steady-state environment, after letting QPFRADJ do its function, you get low values for Active state to ineligible state transitions and a good setting for the activity value. When you have the storage pool with the best activity level value, and understand the dynamics of your environment, consider turning off QPFRADJ (0 value).

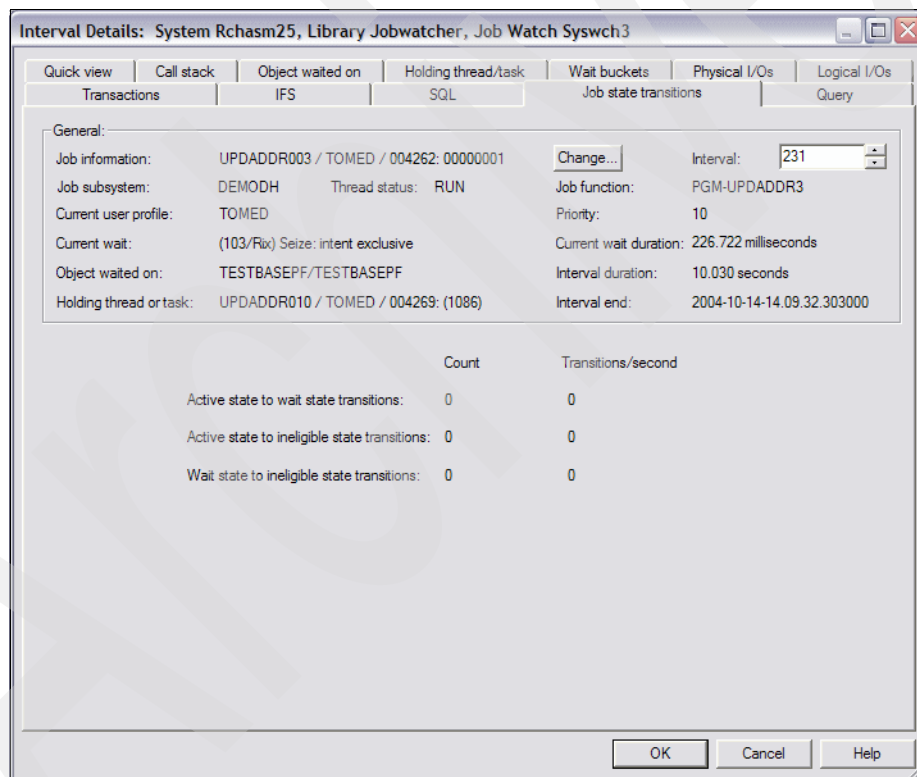


Figure 3-64 Interval Details: Job state transitions tab

► Physical I/Os tab

Along with reads and writes, this tab shows the number of I/O pending faults, page faults, DASD pages allocated and deallocated, and waits for an async write. An example of a wait on an asynchronous write is when we have to wait on a journal bundle to be written out.

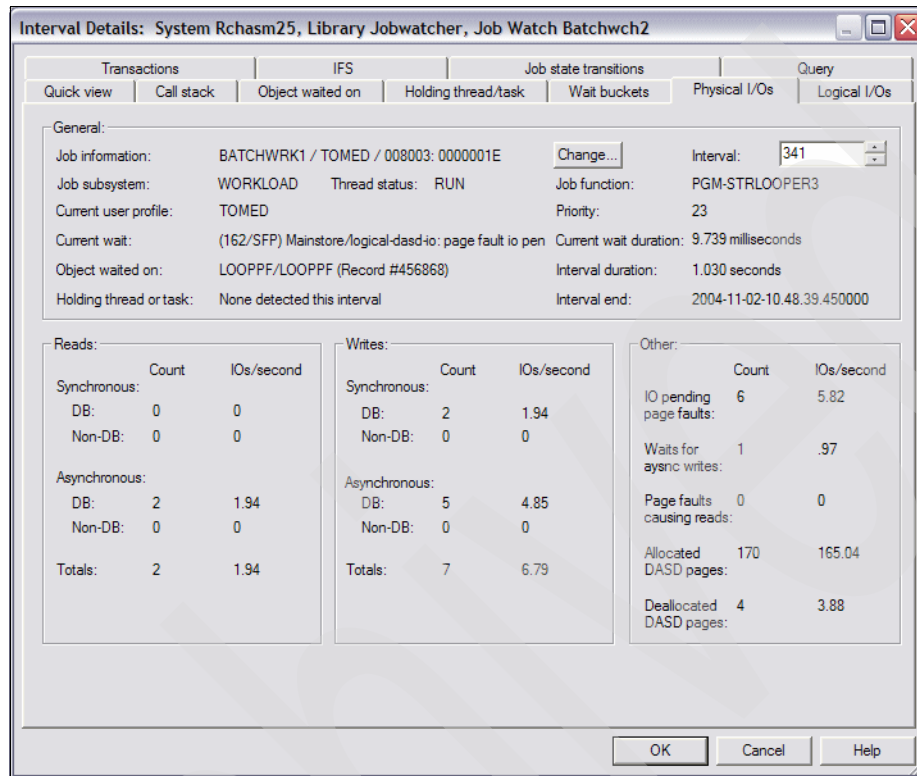


Figure 3-65 Interval Details: Physical I/Os tab

► Logical I/Os

This tab provides a count and per-second rate for read, write, and logical other operations. Logical other includes updates, deletes, commit, rollback, open, close, and other operations not discussed here.

These counts represent each request the application program issues to i5/OS data management. These are commonly referred to as LDIOs.

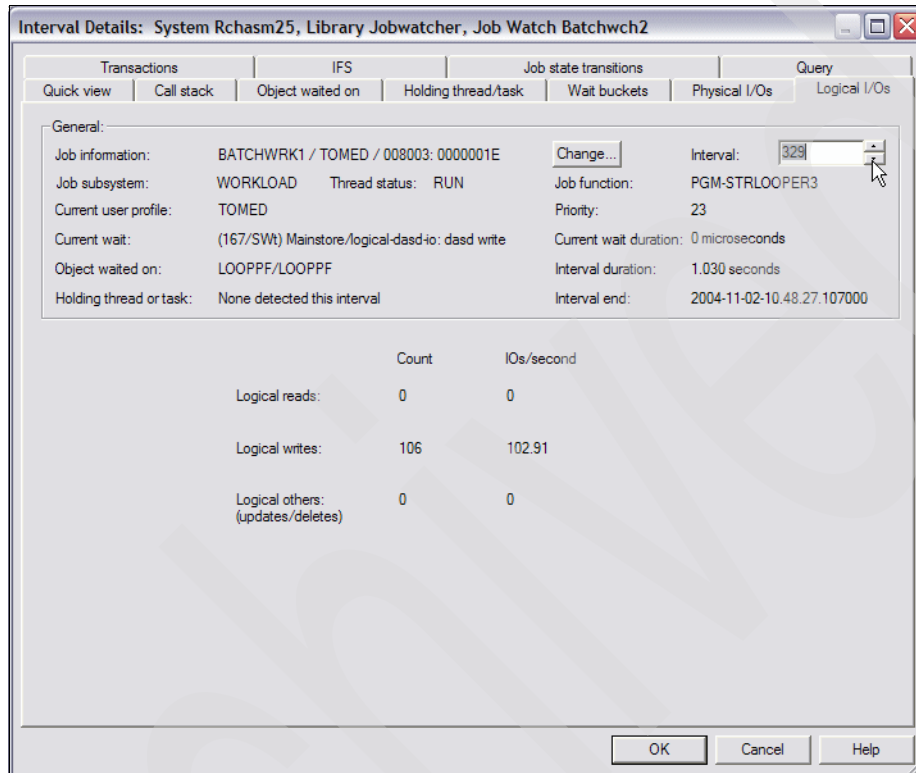


Figure 3-66 Interval Details: Logical I/Os tab

► Query tab

This tab displays the query that was used to build the interval data displayed in the graph. All queries used to build Job Watcher graphs are available and can be copied to a Job Watcher Data Viewer window so that you can modify and build your own queries.

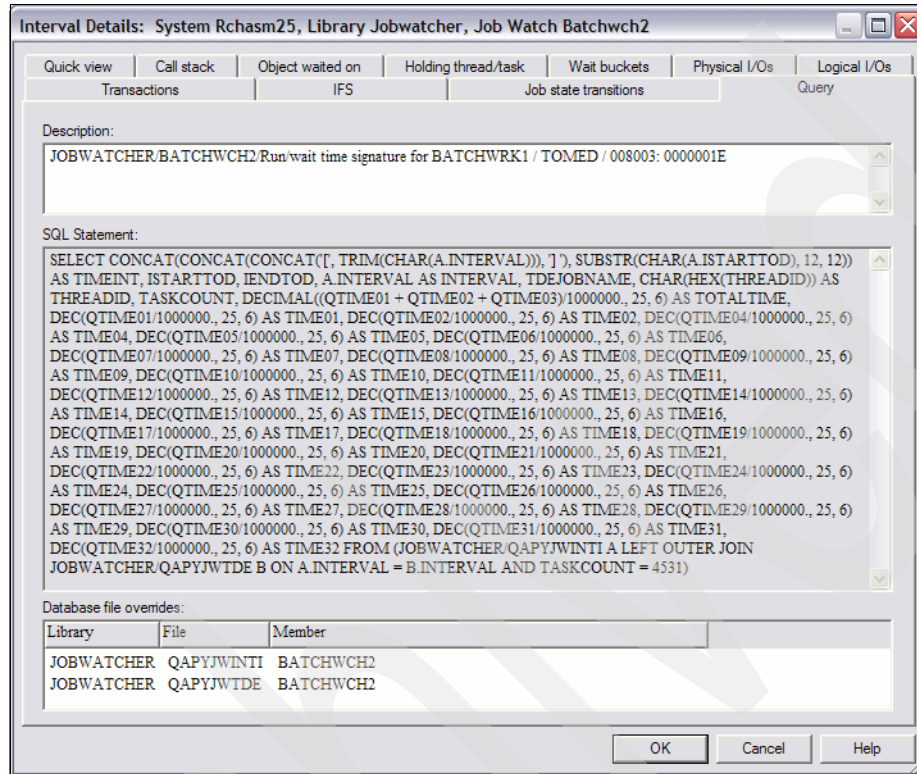


Figure 3-67 Interval Details: Query tab

3.5.6 Additional Job Watcher graphs

More than 50 different graphs are included with Job Watcher. Some display a view by interval and others display jobs ranked in order of time spent in a particular wait condition. In this section we include a sample of the available graphs:

1. Wait graphs by interval: These graphs show an interval-by-interval view displaying the amount of time spent in a particular wait during each interval. We have already seen an example of using a Wait graph by interval in Figure 3-43 on page 86 with a Collection overview signature that shows a number of common waits. There are also graphs that are more specific, such as the Seize and lock time signature seen in Figure 3-68 on page 109 and the DASD time signature in Figure 3-69 on page 109.

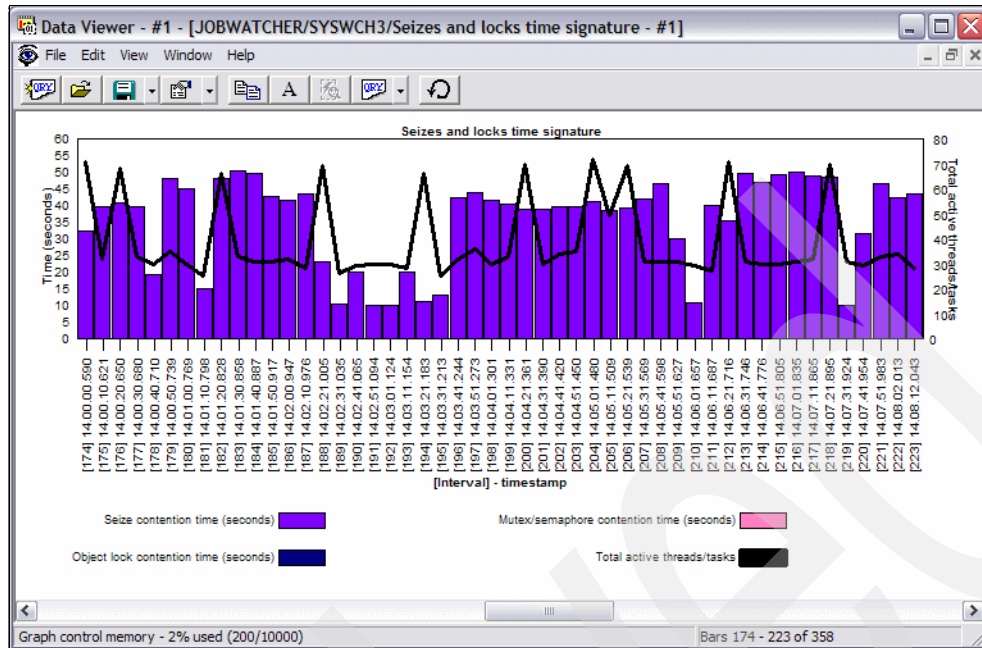


Figure 3-68 Seize and lock time signature

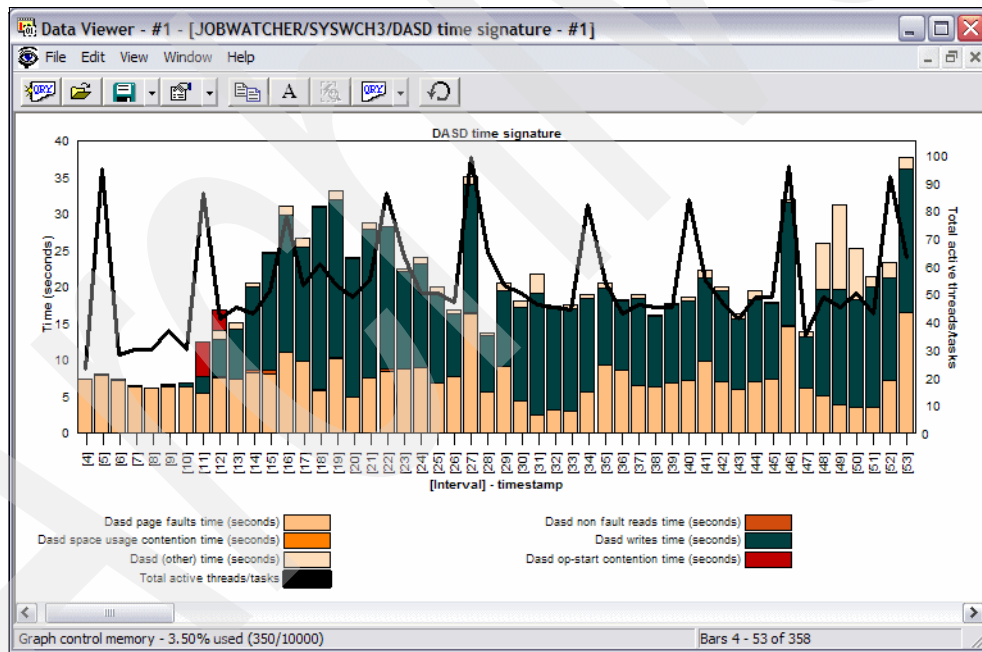


Figure 3-69 DASD time signature

- Wait graphs by job: These graphs display jobs in the order in which they spent in a particular wait condition. Figure 3-70 shows an example of job signatures ranked by time spent in seize contention.

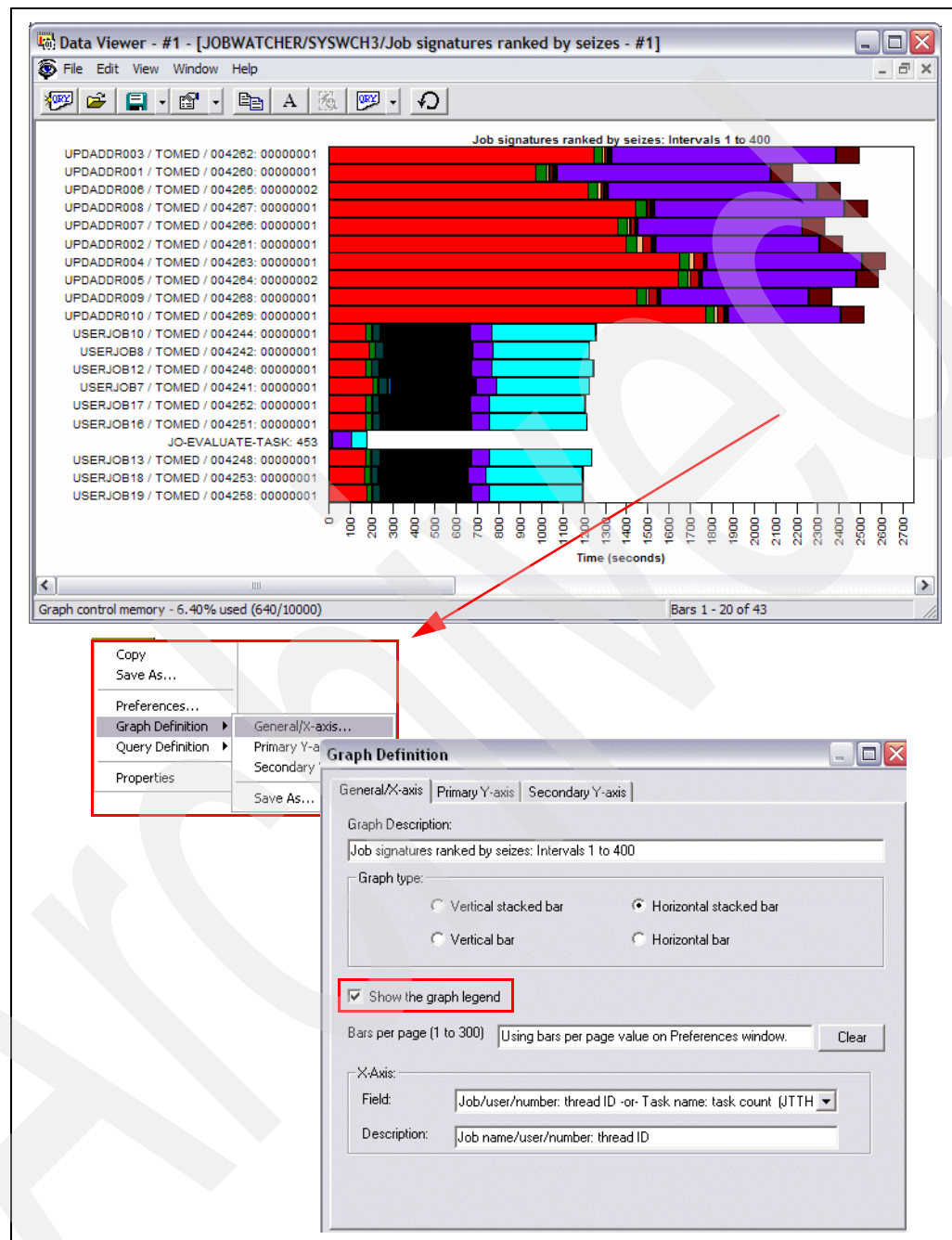


Figure 3-70 Job signatures ranked by seizures

Tip: The Job signatures ranked by seizes (Figure 3-70) and Job signature ranked by CPU (Figure 3-71) graphs default to not showing the performance components color legend. This is because these job signature graphs contain 32 buckets and 32 fields, which means that the legend takes up half of the available window space. However, you can get the color legend to be displayed:

1. Using Figure 3-70 as an example, right-click anywhere in the **Job signatures ranked by seizes** window.
2. In the context menu, select **Graph Definition** → **General/X-axis** as shown.
3. Check the box for **Show the graph legend**. Click **OK** (not shown).

The graph window contains the job bar graphs above the colors legend, as shown in our Job signature ranked by CPU example (Figure 3-71).

We explain more about graph definition options in “Graph views” on page 238.

4. CPU: CPU graphs are very helpful in determining which threads or tasks are using CPU during an entire collection or just during certain intervals. (The description for Figure 3-40 on page 83 includes an Edit → Preferences example of changing the range on which the summarization is done.) Figure 3-71 shows Job signatures ranked by CPU.

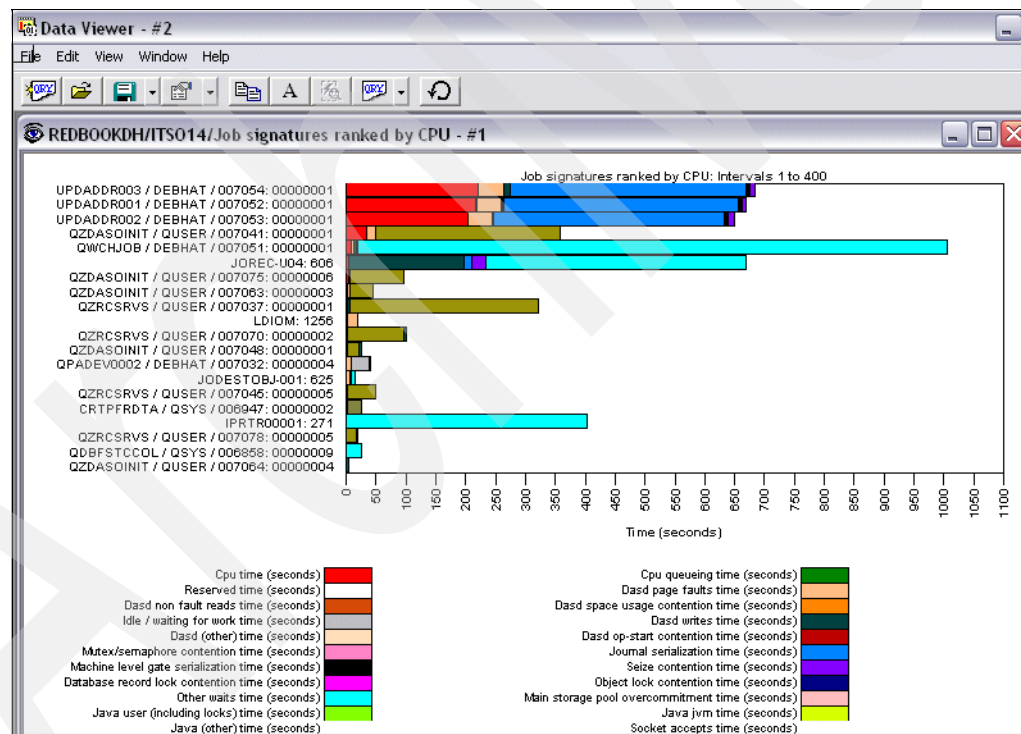


Figure 3-71 Job signature ranked by CPU, showing color legend

- Another useful CPU graph is the CPU/CPUq usage high-priority and low-priority jobs as shown in Figure 3-72.

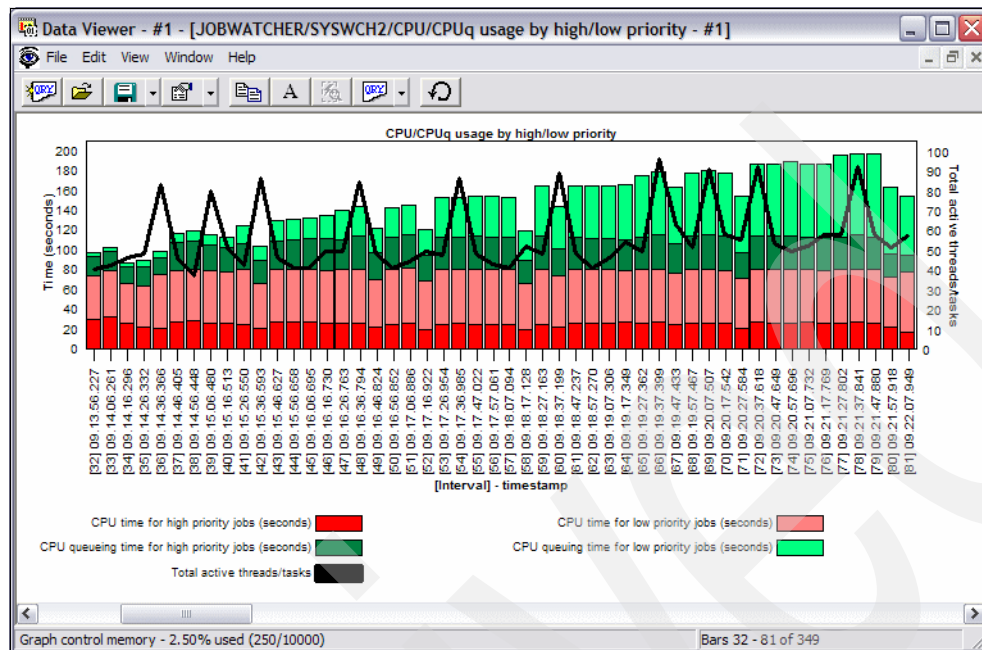


Figure 3-72 CPU/CPUq usage by high/low priority

The CPU/CPUq usage graph is an overview of all threads or tasks that were included in the Job Watcher collection and the amount of time spent either dispatched or queued. The graph also breaks the jobs out by high priority and low priority for a better view of the impact high-priority jobs are having on the low-priority jobs.

- The CPU/CPUq graph is currently configured where high-priority jobs are any job or thread with a priority of 29 or less. This value can be changed using the field selection option of the query definition pull-down menu. See Figure 3-73 and Figure 3-74.

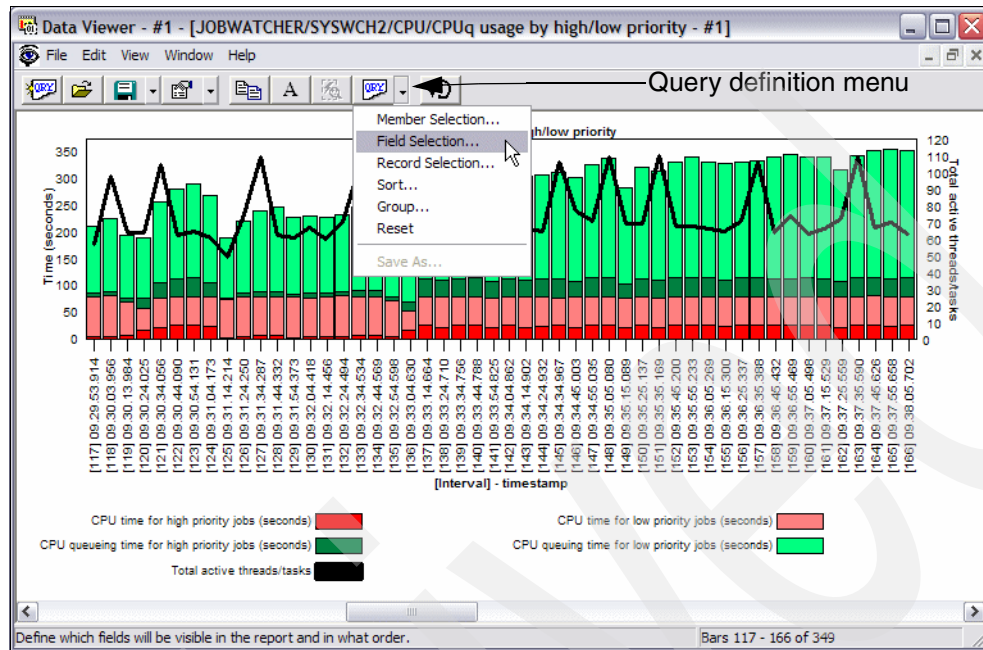


Figure 3-73 Query field selection

Show?	Field Description	Field	SQL Expression
<input checked="" type="checkbox"/>	TIMEINT	TIMEINT	CONCAT(CONCAT(CONCAT(T, TRIM(CHAR(INTERVAL))), 1
<input checked="" type="checkbox"/>	Interval number	INTERVAL	<none>
<input checked="" type="checkbox"/>	CPUHIGH	CPUHIGH	READPRI <= 29 then QTIME01 else 0 end
<input checked="" type="checkbox"/>	CPULOW	CPULOW	DEC(DEC(sum(case when THREADPRI > 29 then QTIME01 e
<input checked="" type="checkbox"/>	CPUQHIG	CPUQHIG	DEC(DEC(sum(case when THREADPRI <= 29 then QTIME02
<input checked="" type="checkbox"/>	CPUQLOW	CPUQLOW	DEC(DEC(sum(case when THREADPRI > 29 then QTIME02 e
<input checked="" type="checkbox"/>	TOTJOBS	TOTJOBS	COUNT(*)
<input type="checkbox"/>	Task count	TASKCOUNT	<none>
<input type="checkbox"/>	Elapsed interval time in microsecs	TDEUSECS	<none>
<input type="checkbox"/>	Time of day at snapshot start	STARTOD	<none>
<input type="checkbox"/>	Microseconds since IPL at snap...	STARTUSECS	<none>
<input type="checkbox"/>	Microseconds since IPL at snap...	ENDUSECS	<none>
<input type="checkbox"/>	Thread ID in hex	THREADID	<none>
<input type="checkbox"/>	Process initial thread task count	ITASKCOUNT	<none>
<input type="checkbox"/>	Job/task name	TDEJOBNAME	<none>
<input type="checkbox"/>	Thread status	THRDSTATUS	<none>
<input type="checkbox"/>	Current user profile	CURRUP	<none>
<input type="checkbox"/>	Job/task birth time of day	BIRTHDAY	<none>

Figure 3-74 Query Definition

In the Query Definition window, click on the SQL expression for fields CPUHIGH, CPULOW, CPUQHIG, and CPUQLOW to change 29 to the desired value in each.

Important - CPU time definition: In all or nearly all of the screen captures and associated text, the label for Run/Wait bucket1 is referred to as CPU. By the time this publication is available, the Job Watcher tool will be referring to bucket1 with the more accurate label of **Time Dispatched on a CPU**. See “Bucket 1: Dispatched Time (previously referred to as CPU)” on page 33 for the definition of Run/Wait bucket1.

7. DASD/ IO graphs: We have included three examples of the graphs available in the DASD/IO folder for an idea of what is available. The Physical I/O requests graph shown in Figure 3-75 is probably the one we use most often when analyzing disk I/Os.

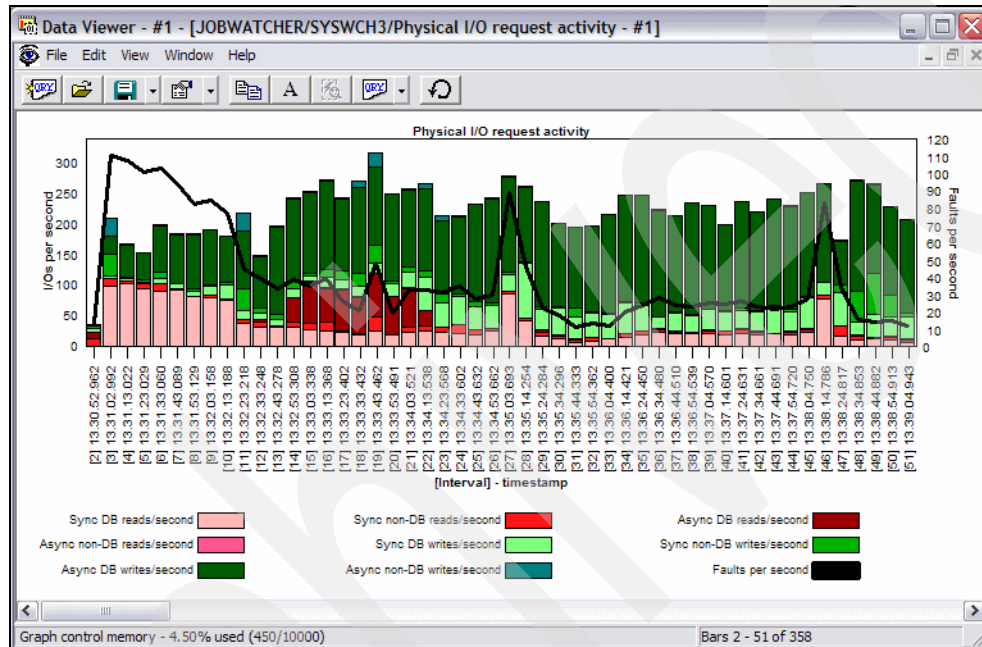


Figure 3-75 Physical I/O request activity

When dealing with a problem that includes busy disk units (for example, disks are averaging more than 30-40% utilization), the Physical I/O request activity graph gives you a view of the type of I/Os that are causing your disk units to be busy. You can also see the breakdown between synchronous and asynchronous I/Os.

8. In the example shown in Figure 3-76, the y axis is the number of active threads/tasks. In this Job Watcher collection, the y axis is a straight line across the top of the graph (A) because we were collecting for only one job.

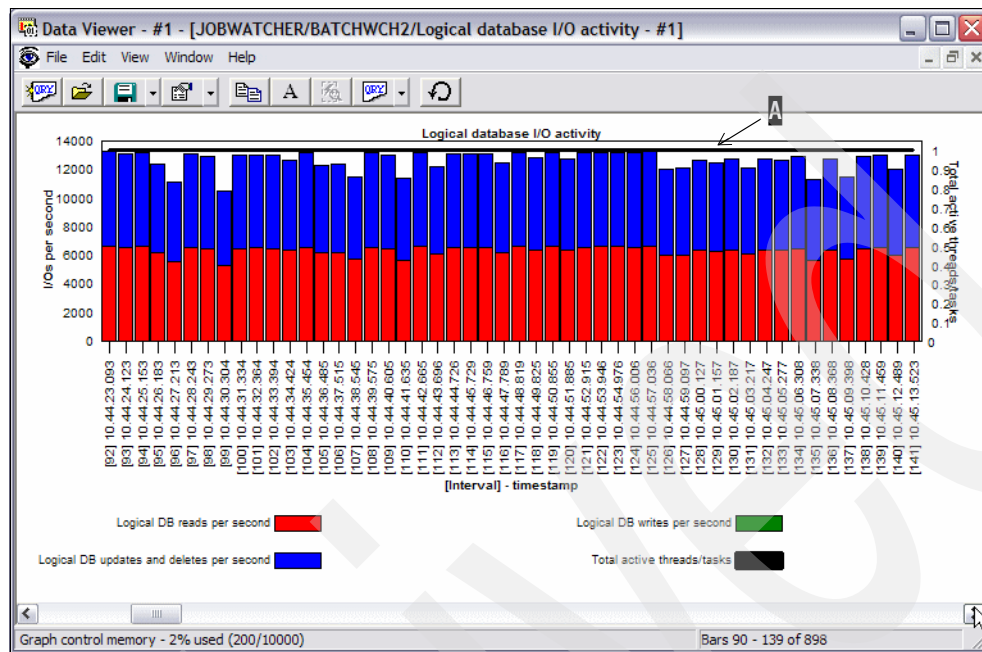


Figure 3-76 Logical database I/O activity

The Page faults graph can help us assess the amount of storage that is available to run applications using the same storage pool. The newer 8xx family of servers and the eServer i5 POWER5 servers, along with the newer technology disk controllers and disk drives themselves, can sustain rather high per-faults-per-second rates.

In the example in Figure 3-77, the peak Faults per second rates are low enough that page faulting (discussed in more detail in “Page faults” on page 146) does not appear to be an indicator of a performance problem.

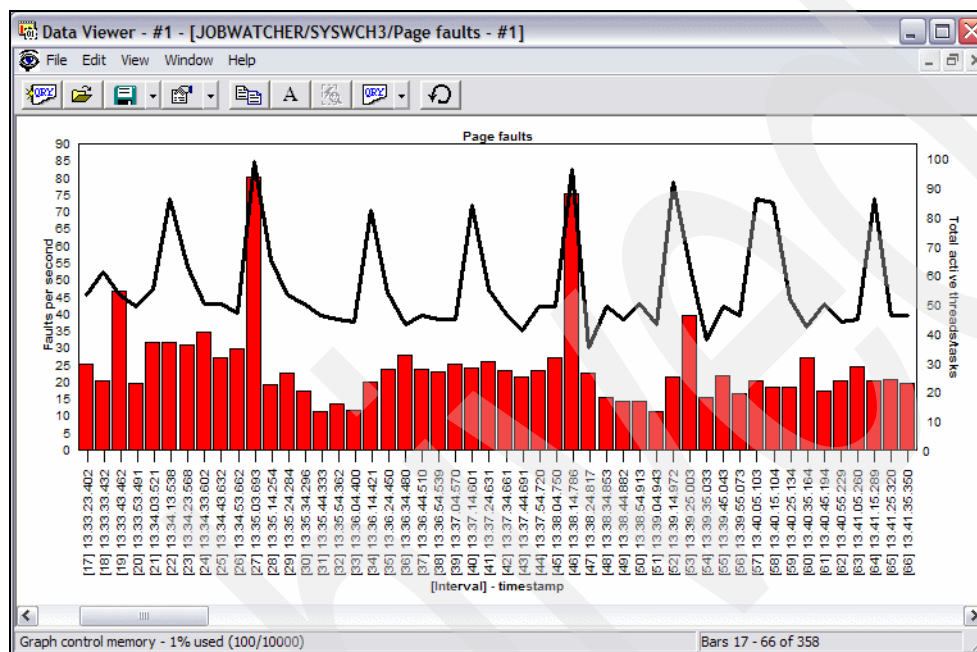


Figure 3-77 Page faults graph

Important: Even with the extensive number of graphs that are available, ultimately you will find that you may have to go to the Job Watcher output data and run your own queries over the data in order to conclude your analysis. In Appendix C, “Querying and graphing tips for Job Watcher” on page 221, we discuss creating user-defined (customized) queries and also include some examples.

Analysis example: traditional batch ILE RPG application

In this chapter we demonstrate how to use Job Watcher to analyze performance problems related to a traditional batch ILE RPG workload.

We use an example of an RPG batch job that is now running longer than it previously did although system administrators feel it does not have additional work to perform.

Important:

- ▶ Remember that Job Watcher is a sample-based collection tool, not a trace-based collection tool.
- ▶ To examine call stack data (used in this example), you must have specified to do call stack data capture when you started Job Watcher data collection.
- ▶ To understand query examples shown in this chapter and write your own queries, you must refer to wait bucket definitions listed in Table 2-3 on page 30 of this chapter and the Job Watcher file and field descriptions included in “Job Watcher terminology” on page 200.
- ▶ This chapter assumes that you are familiar with the contents of Chapter 3, “Getting started” on page 51.

Refer to 3.3.4, “Data collection options” on page 58 for information about specifying data collection options.

4.1 Analyzing the problem

In this RPG traditional batch application, we had Collection Services data both from when the job ran its original run time and from its longer run time. For each job, we created the performance database files and printed the Component Report to examine the number of logical I/O operations and for any other indications. The longer-running job obviously used more CPU during each Collection Services interval and ran over more intervals than the shorter-running job did. Looking at these reports, we saw that the number of logical disk I/Os were close enough between the two jobs to conclude that the longer run time was not due to significantly more records being processed.

But we could not get a feel for what was causing the high CPU usage.

In our example, there could be several reasons for the longer run time, such as more page faulting because someone lowered the pool size storage used by the job, or someone made some “should not affect performance” changes to one or more of the programs within the application job. If there were program changes, one of the causes could be a misuse of file opens, or, in this example, misuse of activation groups.

4.2 Activation group usage

All programs and service programs running on an iSeries are activated within a substructure of a job that is called an activation group. This substructure contains the resources that are necessary to run the programs and procedures, including static and heap storage, and temporary data management resources such as opened file control blocks. These resources can be scoped to a specific activation group. Access to the information within an activation group can be prohibited from applications scoped to a different activation group.

Activation groups offer a powerful function, but too much activation group processing can have a negative impact on performance.

The system provides a default activation group (*DFTACTGRP) within a job that can be used to contain all resources for an entire job. IBM-supplied QLGLOCAL also appears as an activation group when certain system functions are used to access job attributes.

In most application environments, use of the system-supplied activation groups is sufficient.

However, a user program within a job running an application, by design, can create one or more of its own activation groups and isolate resources to one of these specific activation groups.

Normally you would want to minimize the number of activation groups used in each job, as each creation and change between activation groups can affect performance.

Having set the stage for the topic of activation group usage, we start with the subject of a job's increased CPU utilization being detected to give a Job Watcher example of activation group analysis.

An increase in the CPU time that a job uses can be attributed to a number of sources, such as an increase in workload or a longer code path. A longer code path can be caused by, but not limited to, the improper use of activation groups in ILE RPG.

In this section we use Job Watcher to demonstrate how to determine whether the reason for the increase in CPU utilization is a change in the way activation groups are being used within our application.

Figure 4-1 shows a before-and-after comparison of how our batch jobs ran. In the top graph, the batch job ran for a little more than 7 seconds of CPU time. In the lower graph, the same application took 1,650 seconds of CPU time to complete.

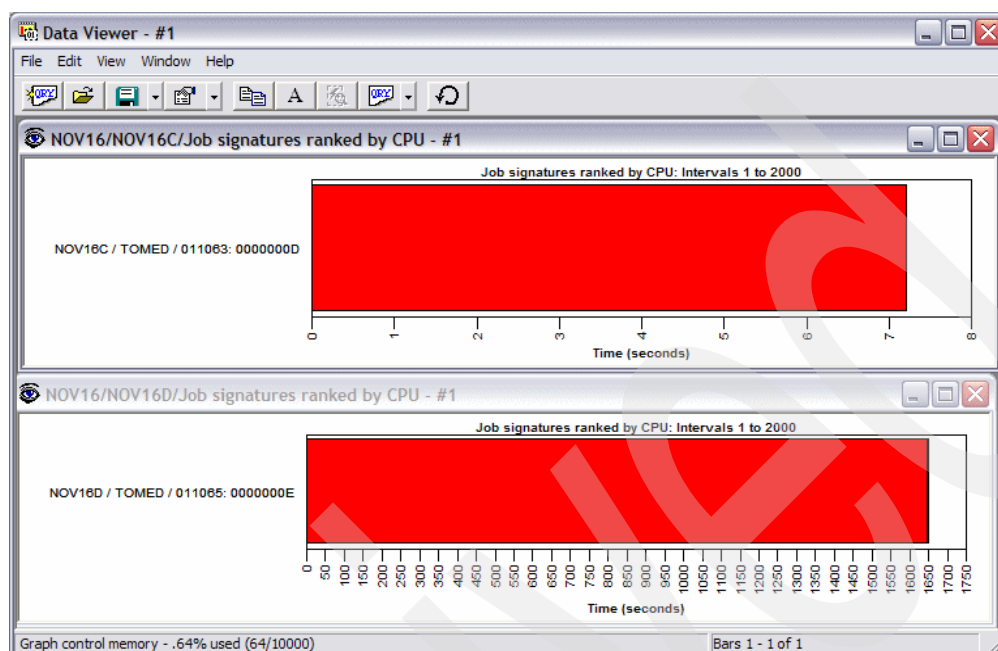


Figure 4-1 Before-and-after comparison

By using the following steps, we determined the cause of the high CPU.

Looking at the call stack of each job is a key way to find out what is going on with the programs running within the job when the snapshot is made. Our Job Watcher collection included call stack information, and the standard Job Watcher GUI can show the basic call stack information for each job within each interval.

We decided we wanted some call stack statistics for each job, such as “viewing all call stacks where program xxxxx occurred.” The base call stack information for deriving these statistics is stored in the Job Watcher collection data, but some of the fields are in a format that is not directly available to the V5R3 Job Watcher GUI interface. For example, some of the important data is in binary format.

To handle this situation through V5R3, Job Watcher provides the command Retrieve Stack Data (RTVSTKDTA), which you must run to convert this binary data into a file/member format that can be processed by a user-written query. We describe the RTVSTKDTA command in “RTVSTKDTA command” on page 250. For this example, follow these steps to get the necessary call stack information.

We performed the following RTVSTKDTA command steps to get the call stack information into a query-ready format:

1. `SBMJOB CMD(QIDRGUI/RTVSTKDTA WCHMBR(NOV16C) WCHLIB(NOV16)
MODTYPE(JW53) RCOUNT(*ALL) TCOUNT(*ALL) STKMBRNAME(NOV16C))
JOB(RVTSTACK)`

`SBMJOB CMD(QIDRGUI/RTVSTKDTA WCHMBR(NOV16D) WCHLIB(NOV16)
MODTYPE(JW53) RCOUNT(*ALL) TCOUNT(*ALL) STKMBRNAME(NOV16D))
JOB(RVTSTACK)`

NOV16C is the short-running job collection and NOV16D is the long-running job collection.

2. We ran the following queries to determine the differences in the program and procedures that were called in the two batch jobs.

Note: In this chapter we use the Job Watcher Data Viewer interface for entering SQL SELECT statements. Find more about this capability in 5.2.3, “Data Viewer” on page 134.

Figure 4-2 is an example of the query we ran over the Job Watcher data for the short-running batch job, and Figure 4-3 on page 121 shows an example of the same query that was run over the Job Watcher data for the longer-running job.

COUNT	Task count	Program	Module	Procedure Name
6	10564			aimach_program_call_portal
6	10564	LOOPCL3	LOOPCL3	_CL_PEP
6	10564			pmlInitiateProcessUnderTarget__Fv
6	10564			aiuser_program_call_portal
6	10564			cblabbranch
6	10564	QCMD		
5	10564	LOOPCL3	LOOPCL3	LOOPCL3
5	10564			userProgramResume__15AiUpcall>
2	10564			***No Procedure Found*** TBT>
2	10564	QCLCLC>		
2	10564			#mnressp
2	10564			#mnrrnl
2	10564			#cfmir
2	10564			findObject__16MnResolveContex>
1	10564			syscall_A_portal
1	10564			syscall_22_portal

Figure 4-2 Short-running example

Attention: Because Job Watcher is a sampling tool and not a trace, occasionally the call stack will not be harvested and, instead of a program or procedure being displayed, you will see ***No Procedure Found***.

When a job is running in a processor, Job Watcher will not interrupt the job to get the job’s call stack. This illustrates what we mean by Job Watcher being *nonintrusive*. A true trace would record everything, but would also take increasingly more CPU as the normal application activity increases.

Data Viewer - #1 - [Rchasm25: SQL Query View - #1]

File Edit View Window Help

SELECT count(*) AS COUNT, a.taskcount, a.fpgm, a.fmod, a.fprocname
FROM (SELECT interval, taskcount, fpgm, fmod, fprocname
FROM nov16/qaidrjwstk
GROUP BY interval, taskcount, fpgm, fmod, fprocname) a
GROUP BY taskcount, fpgm, fmod, fprocname
ORDER BY 1 desc

COUNT	Task count	Program	Module	Procedure Name
780	10567			***No Procedure Found*** TBTADDR=0000000000000000
495	10567			cblabbranch
491	10567	LOOPCL	LOOPCL	_CL_PEP
491	10567			aiuser_program_call_portal
491	10567			aimach_program_call_portal
491	10567	LOOPCL	LOOPCL	LOOPCL
491	10567	QCMD		
422	10567			pmInitiateProcessUnderTarget_Fv
317	10567			userProgramCall__15AiUpcallProgramFv
315	10567			pActivatePgm__15AiUpcallProgramFP5AiPgmP9AiProcessR14AilcbClear/
314	10567			targetProgramActivation__15AiUpcallProgramFR14AilcbClearAreaP16AiM
314	10567			ActivateNew__11AiActivatorFv
307	10567			pCreateStdActivation__11AiActivatorFv
168	10567	LOOP	LOOP	_QRNP_PEP_LOOP
148	10567			pBuildDepGraph__11AiActivatorFP5AiPgm
142	10567	QLEAWI	QLEPM	Q LE leActivationInit
126	10567	QLEDAGE	QLEDAGE	_CXX_PEP_Fv
126	10567	QLEDAGE	QLEDAGE	main
124	10567	QLEDAGE	QLEIT	activationInit
110	10567	QLEDAGE	QLEIT	act_Create_FUL
104	10567			smasmswitchstackandcall

Records 1 - 21 of 347

Figure 4-3 Long-running example

In Figure 4-3, notice the difference in the programs and procedures that were in the call stack and the number of times these were called. From this we can determine that a different code path is being used in the longer running job. Why?

The additional programs we see in Figure 4-3 are system programs starting with QLE, which tells us that they are part of the ILE component. We can also see a reference to a procedure called pCreateStdActivation. Even when we do not know anything about LIC procedure names, we can certainly guess the meaning of this procedure.

3. With a very good suspicion that the long-running batch job was creating lots of activation groups, we ran a query over Job Watcher's activation group information file (QAPYJWAIGP) and joined this with the main TDE scope file (QAPYJWTE) to display the name of the activation groups being created along with the job name.

Example 4-1 shows the SQL statement we ran. Example 4-2 on page 122 shows the results for NOV16C (quick-running job), and Example 4-3 on page 122 shows the results for NOV16D (long-running job).

Example 4-1 SQL example for quick-running job

```
SELECT a.totrec, a.taskcount, b.tdejobname, a.actgrpnm
FROM
  (SELECT count(*) as totrec, taskcount,
   case when(actgrp = ' ' ) then '*NEW' else actgrp end as actgrpnm
  FROM nov16/qapyjwaigp
  WHERE substr(actgrp,1,4) <> '*DFT'
  GROUP BY taskcount, actgrp
  )a
LEFT OUTER JOIN
  (SELECT tdejobname, taskcount FROM qapyjwte
```

```

GROUP BY tdejobname, taskcount
) b
on a.taskcount = b.taskcount

```

Example 4-2 Output from the quick-running batch job

TOTREC	Initial thread task count	Job/task name			ACTGRPNAM
7	10,564	NOV16C	TOMED	011063	XXNAMEDXX
7	10,564	NOV16C	TOMED	011063	QLGLOCAL

Example 4-3 Output from the long-running batch job

TOTREC	Initial thread task count	Job/task name			ACTGRPNAM
906	10,567	NOV16D	TOMED	011065	*NEW
1,009	10,567	NOV16D	TOMED	011065	QLGLOCAL

You can see under the heading TOTREC in the longer-running job (Example 4-3) that the management of activation groups has increased significantly. Many more activation groups are being created, and 906 of them have the name *NEW. When a new activation group name is created, every time a program is called it results in the activation group name field in the QAPYJWAIGP file being blank. This is why in our query (Example 4-1 on page 121) we replace the blank with *NEW to show that these are new activation groups.

4. We then used the operating system Display Program (DSPPGM) command to show what type of activation group is used. See Example 4-4 for the DSPPGM command output example of the program we used in our batch job. If you see that the activation group is set to *CALLER, then run DSPPGM for the calling program to determine what it is set to.

Example 4-4 DSPPGM PGM(WORKLOAD/LOOP)

Program :	LOOP	Library :	WORKLOAD
Owner :	TOMED		
Program attribute . . :	RPGLE		
Detail :	*BASIC		
Program creation information:			
Program creation date/time :	11/16/04	11:58:30	
Type of program :	ILE		
Program entry procedure module :	LOOP		
Library :	QTEMP		
Activation group attribute :	*NEW		
Shared activation group :	*NO		
User profile :	*USER		
Use adopted authority :	*YES		
Coded character set identifier :	65535		
Number of modules :	1		
			More..

By using the DSPPGM command on several programs running in our batch job, we found which one (LOOP, in our case) was causing new activation groups to be created. In a more complex application, you may have several programs using new activation groups.

Tip: For ILE performance, have your driver program compiled to use the system-supplied *DFTACTGRP activation group or use a named activation group and then have all called programs from that driver program compiled to use *CALLER.

Excellent sources for more information about developing iSeries applications using the Integrated Language Environment® (ILE) facilities include:

► iSeries Information Center Web site:

<http://www.ibm.com/eserver/iseries/infocenter>

- Select the appropriate region and language.
- In the left navigation pane, expand **Programming** → **Languages** → **ILE Concepts**.

► ITSO Redbooks:

<http://www.redbooks.ibm.com>

- Search with **Integrated AND RPG**.

Archived

SQL, call stack, and journal analysis examples

The chapter demonstrates how Job Watcher can help answer the following questions:

- ▶ Is there any long-running SQL on the system that merits further investigation?
- ▶ What are the most popular SQL statements being used?
- ▶ What application and system programs are on the call stack when SQL is being run?
- ▶ Are there any journal delays? What could be done to reduce them?

This chapter discusses how to drill down through Job Watcher data to find out what is happening on a system running an SQL and journaling environment. We also examine call stack information. This chapter also provides several SQL statement examples showing Job Watcher information in addition to that which is directly available from Job Watcher graphs.

Important:

- ▶ Remember that Job Watcher is a sample-based collection tool and not a trace-based collection tool. Depending on your collection interval value and the run time of your SQL statements, you may miss some SQL statements that have run during the collection.
- ▶ To examine SQL information, you must have specified SQL statement data capture when you start the Job Watcher data collection.
- ▶ To examine call stack data you must have specified to do call stack data capture when you start Job Watcher data collection.
- ▶ To understand query examples shown in this chapter and write your own queries, refer to wait bucket definitions listed in Table 2-3 on page 30 of this chapter and the Job Watcher file and field descriptions included in “Job Watcher terminology” on page 200.
- ▶ This chapter assumes that you are familiar with the contents of Chapter 3, “Getting started” on page 51.

Refer to “Data collection options” on page 58 for information about specifying data collection options.

5.1 Our example application and preliminary performance analysis

Our example application performs SQL-based adds (INSERTs) and updates to an arrival sequence physical file called TESTBASEPF. There are 20 keyed logical files over this physical file, all with immediately maintained access paths. The 20 logical files are called TESTLF01 through TESTLF20.

This application is used for all of the analyses in this chapter.

The application submits three jobs to a job queue; the names of these jobs all begin with UPDADDR. One of these jobs adds 1000 records to the physical file TESTBASEPF. All three jobs update approximately one-third of the physical file TESTBASEPF using SQL UPDATE. File TESTBASEPF has 250,000 records. The updates to file TESTBASEPF affect all of the immediately maintained access paths of the 20 logical files. The physical file and its associated logical files are being journaled with before and after images to a journal called DEMODHJRN. QAPYJWSTS.

Prior to collecting Job Watcher data for our sample application, we took the following actions:

1. We ran Collection Services and looked at the collected data through the System and Component reports, and the display performance data through the DSPFRDTA command and the iSeries Navigator performance graphics built-in. We focused on the run time of the SQL statements.
2. We also ran the iSeries Navigator SQL Performance Monitors and looked at the available reports. This helped us determine the Job Watcher functions used in this example. Both the SQL Performance Monitors and Database Monitor are mentioned in this chapter.
3. We determined to run a Job Watcher collection that included capturing SQL information and performed the GUI-related analysis described in this chapter. The following steps describe the setup and running of the Job Watcher collection.
4. We restored the physical file (PF) and 20 logical files (LF) so the application always runs over 250,000 records in TESTBASEPF.
5. The TESTBASEPF physical file and the 20 TESTLF01 through TESTLF20 logical files were purged from memory using SETOBJACC *PURGE.
6. Journaling of PF and LF before-and-after images was ended.
7. Both the journal and its receivers were deleted and recreated.
8. The three UPDADDR* jobs were submitted to a held job queue.
9. Job Watcher system-wide collection was started, using 5-second intervals, callstack data for every interval, and active SQL statements.
10. The three UPDADDR* jobs were released and the jobs ran to completion.
11. The collection was ended.

Attention: In addition to using performance tools such as Job Watcher, as part of your SQL performance analysis you should have the following performance tips PDF available for your reference. It can be found at the iSeries V5R3 Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm>

Select **Printable PDFs and manuals**. Search the table for **performance**. The title is *DB2 UDB for iSeries Database Performance and Query Optimization*.

5.2 SQL analysis

When making the first collection for assessment, we used the following data collection options to ensure that you collect only those SQL statements that are active when the collection snapshots are made. We specified the following collection options, referring to 3.3.4, “Data collection options” on page 58 for configuring these options:

- ▶ Collection interval: 5 seconds
- ▶ Call stack every interval
- ▶ Active SQL statements and host variables every interval
Remember, the SQL statements will be collected only if they are executing when the collection snapshot is taken.
- ▶ System-wide collection (all jobs, all tasks)
- ▶ Set the maximum size to 4,000 MB (assuming you have enough space)
- ▶ Set the number of intervals to 240 (20 minutes)

The SQL collection option that was used was *Active SQL cursors and host variables*. Only SQL statements that are active when the snapshot is run (end of interval) are collected. The last executed SQL statement option was deliberately not chosen as it would cause the most recently run SQL statement to be harvested irrespective of whether it completed 2 hours ago or 2 seconds ago. Hence collection of the last executed SQL statement is not recommended for general system level analysis unless you are already familiar with the applications being run on the system and have prior knowledge of potential long-running SQL statements. One way to do this is to have previously used such tools as the Database Monitor or iSeries Navigator SQL Performance Monitor.

5.2.1 Getting started

Whenever an active SQL statement was collected, it can be seen complete with the host variable values reconstituted using the client workstation GUI. Every collected SQL statement is stored against the interval and task count that ran it; for this reason you must drill down through the GUI graphs until you are viewing data for a single interval and single task count.

We show the most commonly used Job Watcher drill-down techniques in 3.5.3, “Reports and graphs: Example 1” on page 85 and 3.5.4, “Graphs and reports: Example 2” on page 94.

We collected data over the mini-application described in 5.1, “Our example application and preliminary performance analysis” on page 126 into a collection we named ITSO14 in the REDBOOKDH library. The high availability journal performance feature #42 for i5/OS (journal caching) was not used in this example.

We selected the **Wait graphs by interval** → **Collection overview time signature** graph for our collection, as shown in Figure 5-1 on page 128.

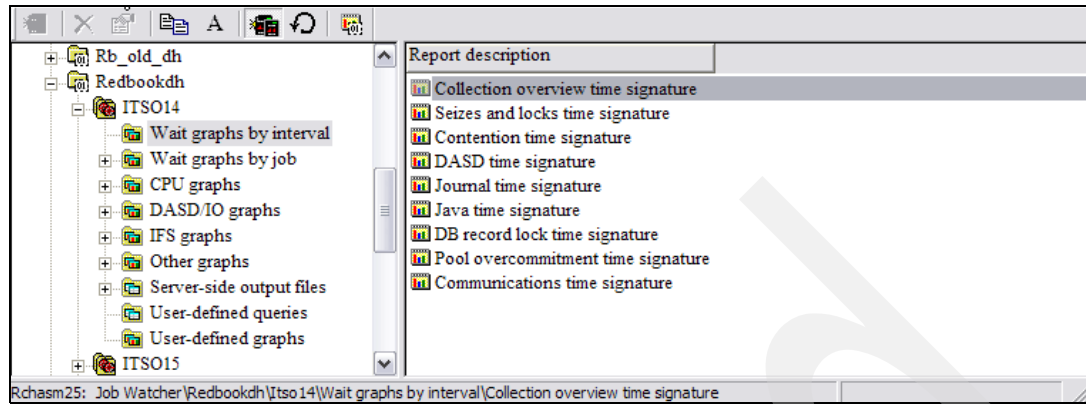


Figure 5-1 Viewing the ITSO14 collection run/wait profile

This produced a collection-wide run/wait graph, by interval, for all jobs and tasks (Figure 5-2).

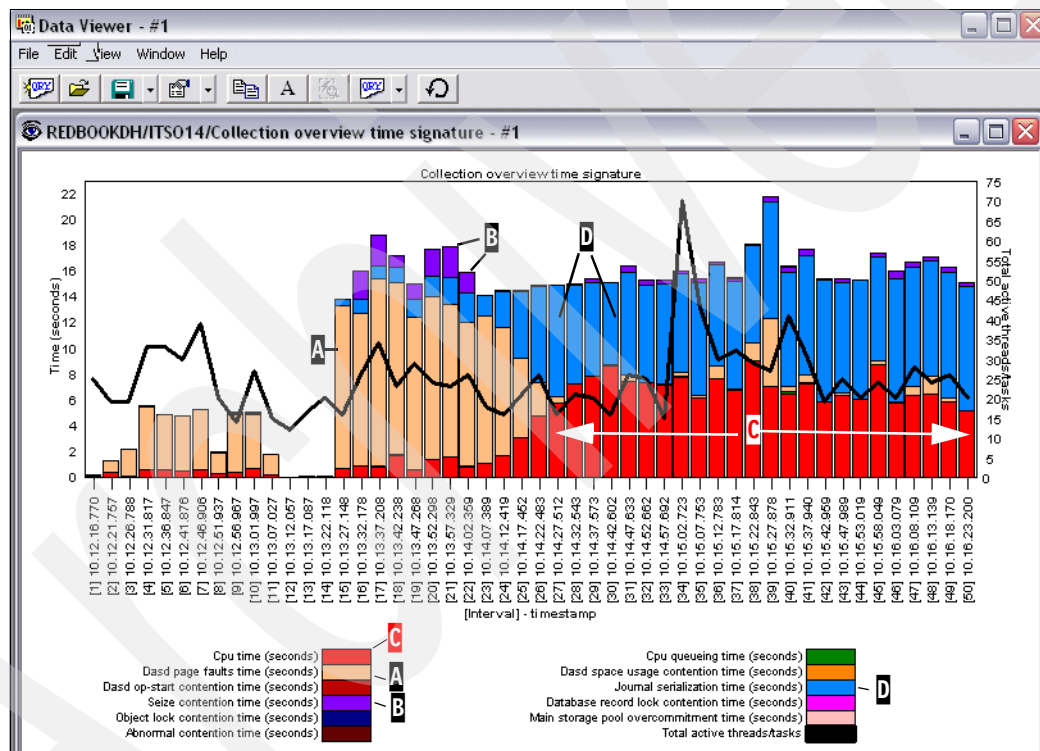


Figure 5-2 Collection overview time signature graph

This graph shows little activity until interval 15, where there was a significant increase in the time spent in page faulting (A). This interval is actually where our example SQL application started. See 5.1, "Our example application and preliminary performance analysis" on page 126, for details of the application, and "Page faults" on page 146 for finding which objects were being faulted into memory.

Attention: Previously, we recommended that you view the graphics in this book online to distinguish the many different colors used for each performance metric. Here, we have also annotated a few of the colors for metrics with reasonably high values to help you see the larger contributors to performance:

- ▶ (B): Seize contention time (purple)
- ▶ (C): CPU time (red)
- ▶ (D): Journal serialization time (blue)

Also note the height of the time interval bars. For each Job Watcher graph function, Job Watcher has a unique default set of wait types (buckets). That one type of graph might include 14 wait buckets and another all 32. The bar heights may also vary based on the number of actual running threads/tasks and idle threads/tasks during each interval.

In this topic we focus on finding which jobs were running the most active SQL statements during the collection samples; these are the jobs that had in-progress SQL statements when the snapshots were taken. We also identify the most popular SQL statements.

5.2.2 Viewing an active SQL statement using the GUI

SQL statements with resolved host variables and SQL PREPARE statement values. You can see the resolved host variables using the GUI only *when you are within a single interval and a single task count*.

Attention: Queries run against the QAPYJWSQL file will show SQL statements, but without host variables and prepare values resolved.

We start by seeing whether there were any active SQL statements in progress in interval 15, as this is the first interval in which any significant work started. To be able to check for SQL statement presence, we must have drilled down to a single task count/interval level. We first examine the jobs (task counts) running in interval 15, then pick a job to check for SQL statements. As most of the time in interval 15 was spent on DASD page faults, right-click in the DASD page faults stacked-bar entry for interval 15 (see Figure 5-3 on page 130). This defaults to producing a list of active jobs contributing to these faults (Figure 5-4 on page 130).

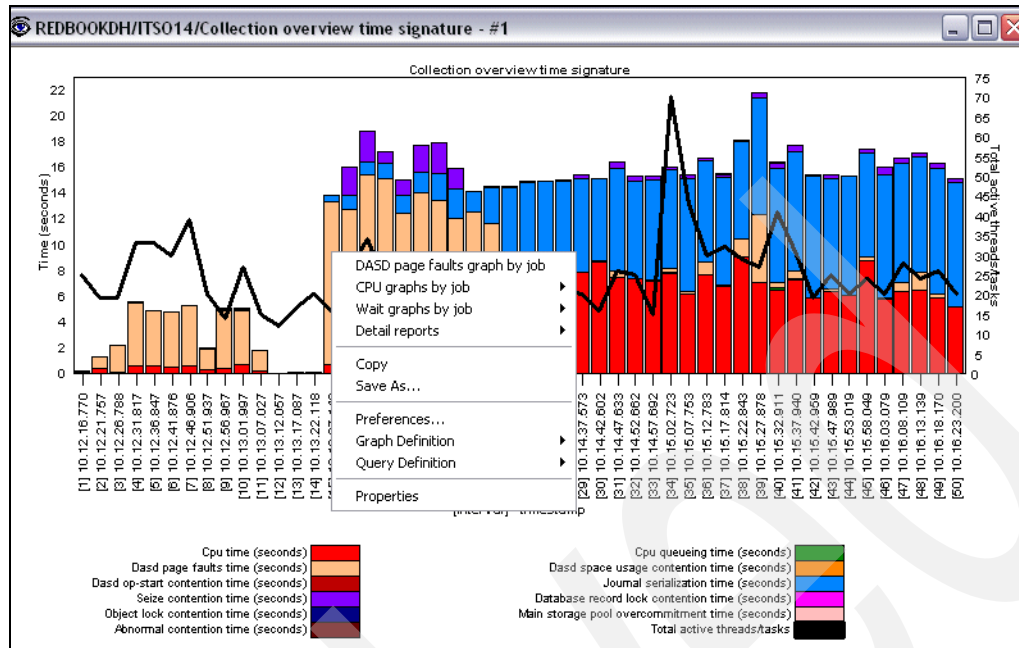


Figure 5-3 How to find the jobs in interval 15

One or more task counts in these jobs used more than 0 CPU in this interval and all of the jobs shown contributed more than 0 seconds to the DASD page fault total during the interval.

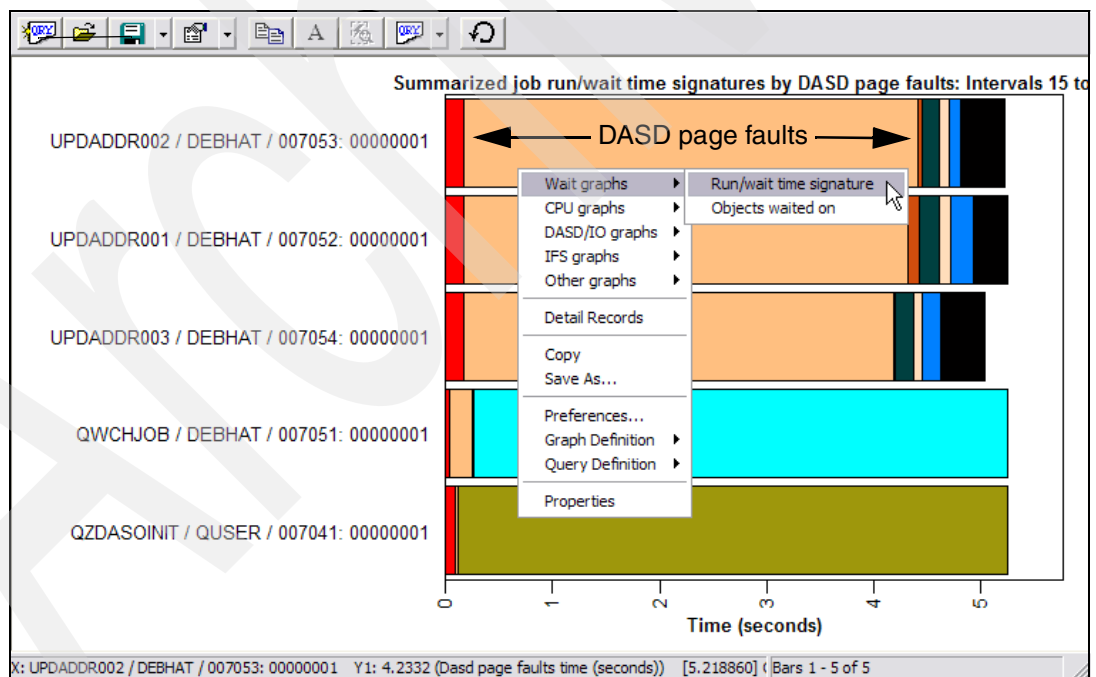


Figure 5-4 The jobs causing DASD page faults in interval 15

To see the run/wait profile for the job UPDADDR002 for all intervals (Figure 5-5), right-click in the job and select **Wait graphs** → **Run/wait time signature** (as shown in Figure 5-4 on page 130).

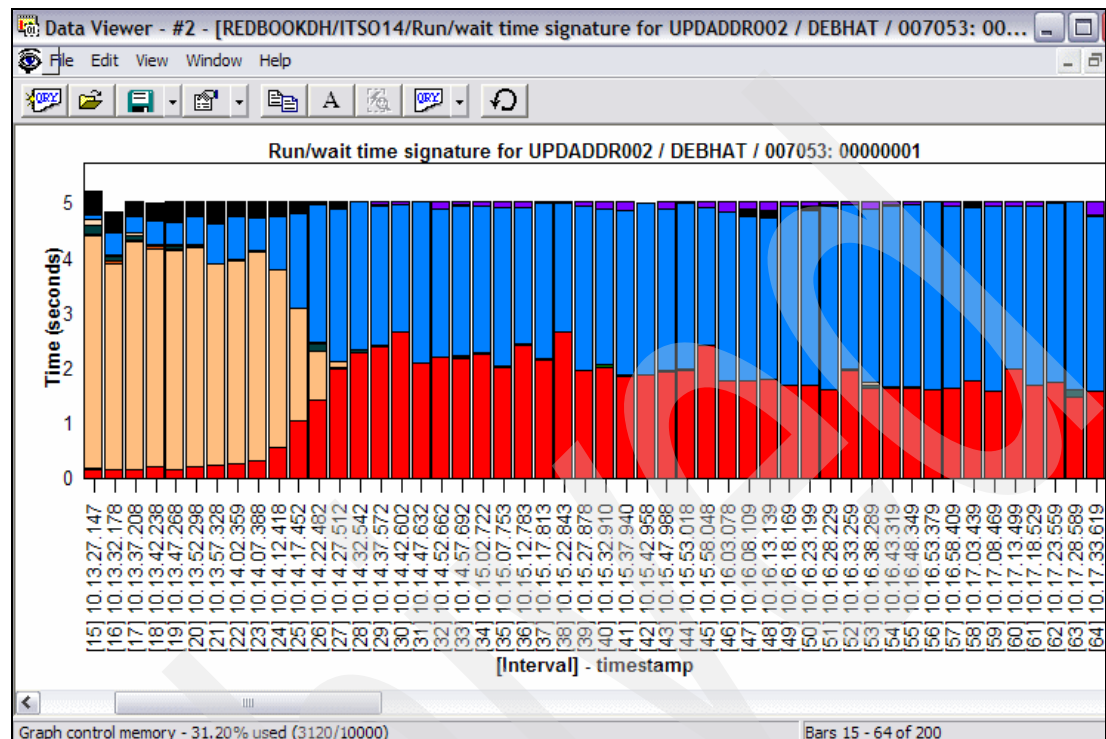


Figure 5-5 Run/wait signature by interval for one job

In this example, the page faulting appears significant only as the job starts up. We are considering this as normal and not part of our problem determination effort. The page faults are probably file/table open and work area creation that is necessary for all jobs.

If significant page faults showed up throughout the job then we would investigate whether file closes and opens are being repeated or perhaps some action reduced the size of the storage pool this job was running in.

Call stack analysis, which we include in our SQL statement run time example, would help identify programs being called repetitively, such as calls to operating system file close and open functions that could be one of the causes of poor performance.

We now drill down within the details for the job UPDADDR002, for all intervals in the collection, but do not focus on the cause of the page faulting.

We do, however, start with interval 15 in order to see the interval details that could include SQL statement and call stack details. Again, remember that interval details are only available when you have selected one job's task count and one interval to inspect. We clicked in interval 15 to open the Interval Details window shown in Figure 5-6 on page 132, which is open to the Quick view of job details.

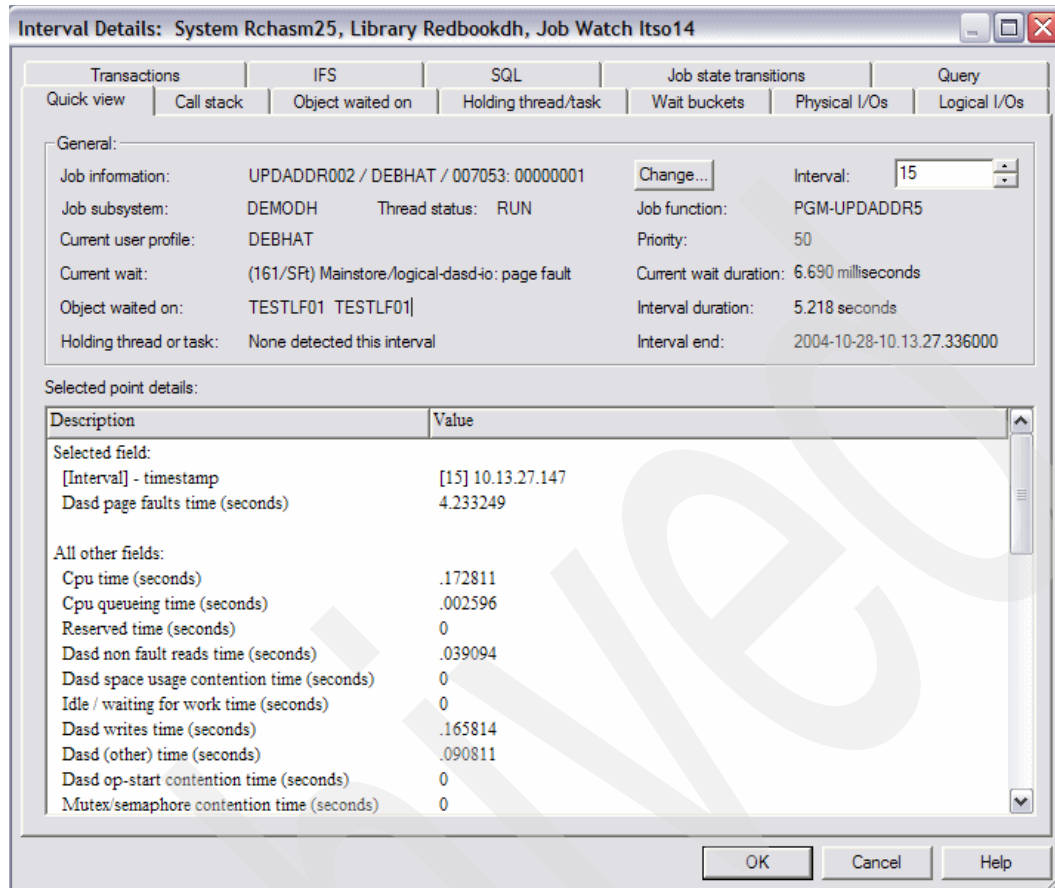


Figure 5-6 Interval details for one task count in one interval

Click the **SQL** tab to see the resolved SQL statement as shown in Figure 5-7 on page 133.

There is either one or no SQL statement per task count per interval. This Interval Details window is the only place in Job Watcher where the SQL statement is shown.

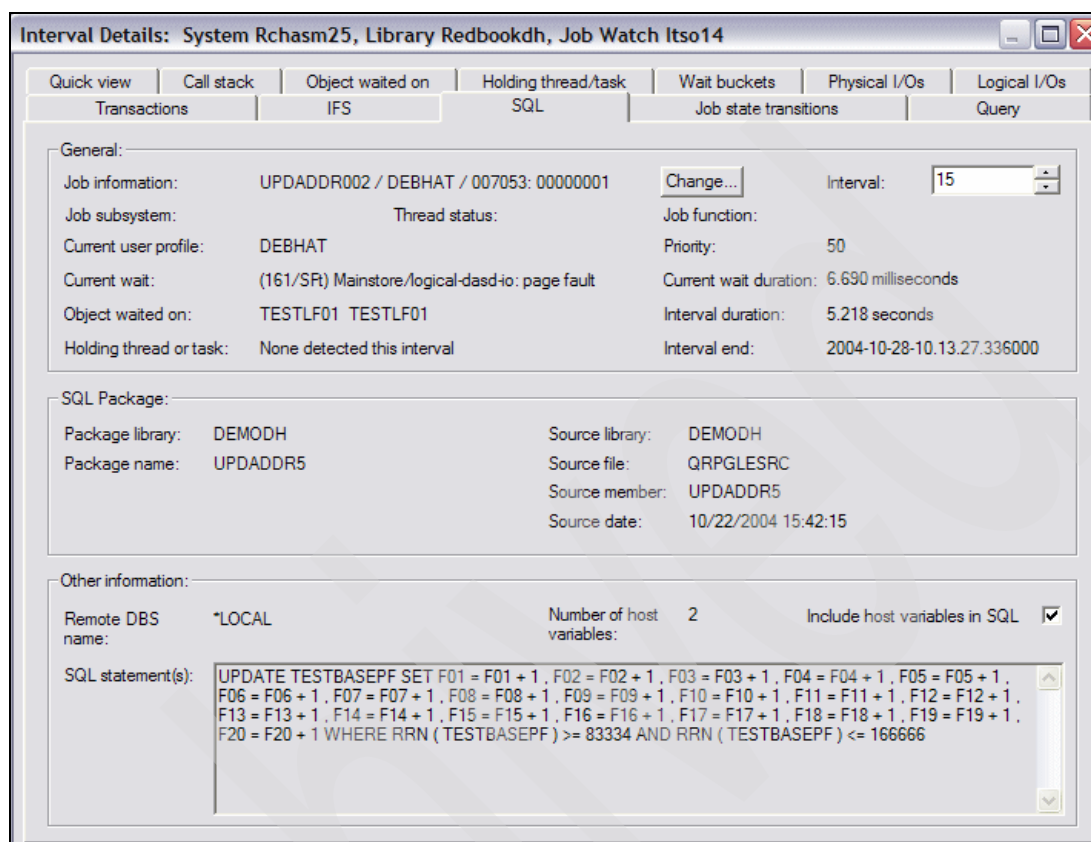


Figure 5-7 SQL statement detail, with host variable values resolved

The unresolved SQL statements are stored in the QAPYJWSQL file, with the host variables stored in the QAPYJWSQLH file. The reconstitution of the host variables (fields or columns whose values can change for each execution of the SQL statement) back into the SQL statement is only provided via the GUI. In this example, the host variable is RRN.

At the time this book was published, there was no Job Watcher command available to reconstitute the host variables for all SQL statements for all task counts into a file.

If you are familiar with SQL, after you have seen the type of SQL statements being run you may want to collect DB Monitor data or run the iSeries Navigator SQL Performance Monitor over one or more jobs to determine whether any SQL performance tuning should be made. These tools help identify whether creating an index over the tables used by the SQL statement could improve performance.

Note that in some i5/OS documentation, an index is sometimes also referred to as an *access path* or *logical file*.

We next discuss the Job Watcher Data Viewer interface, as using the Data View is frequently necessary in order to run queries that find SQL statements.

5.2.3 Data Viewer

This is an environment provided as part of Job Watcher that enables you to run your own SQL-based SELECT statement queries to gather more information from the Job Watcher data. You are restricted to running only SQL SELECT statements and SELECT must begin the SQL statement. SQL CREATE table, CREATE INDEX, DROP TABLE, and DELETE cannot be run in this environment.

If you do not know SQL, the Data Viewer offers a *Query definition* interface that helps you build a moderately complex SQL statement to process the collected Job Watcher SQL data. However, this interface restricts you to working with only one file.

To run more complex SQL-based queries, the Data Viewer offers an *SQL Query view* where you can manually enter very sophisticated (full-function) SQL SELECT statements. This interface enables you to work with more than one file, using JOINS or UNIONS. However, you need to know how to write SQL SELECT syntax as there is no assistance with building the statement in this environment.

See Appendix C, “Querying and graphing tips for Job Watcher” on page 221 for the most complete description of Data Viewer and how to use either the Query definition interface (for novice SQL users) or the SQL Query view (for expert SQL users) interface and the functions available within each interface.

All of the queries in 5.2.4, “Finding jobs running most SQL statements” on page 134 through 5.3.4, “Finding any SQL-related or database-related system activity” on page 143 were produced using the full-function Data Viewer SQL Query view interface.

5.2.4 Finding jobs running most SQL statements

We need to check what jobs were found running active SQL statements during the collection.

The first action we take is to find the total active SQL statement count for the collection:

1. Open a new Data Viewer by right-clicking **Job Watcher** and selecting **Open New Data Viewer** as shown in Figure 5-8.

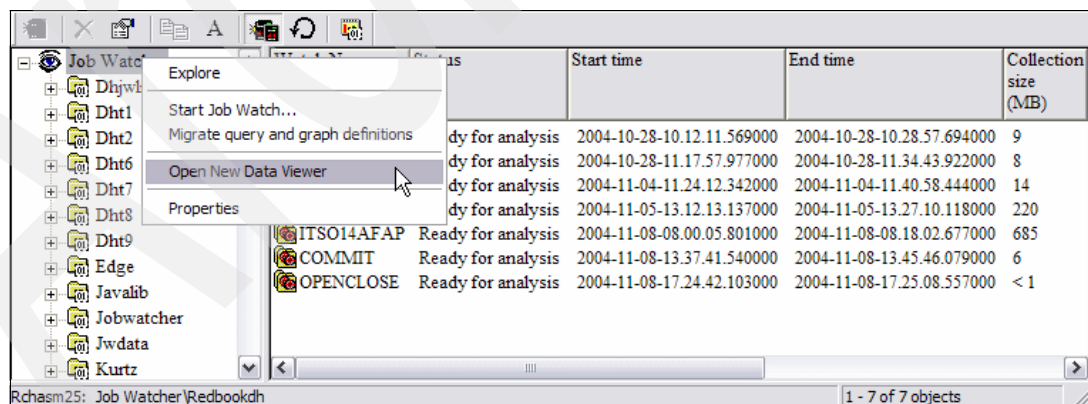


Figure 5-8 Starting a new Data Viewer

2. In the new Data Viewer, enter the SQL statement exactly as shown in Figure 5-9.

This statement counts the number of records in the QAPYJWSQL file collection member that have an SQL statement length of greater than zero (>0). When a sample snapshot runs, a task count is either running active SQL or not. There is one record in the QPAYJWSQL file in each interval for every task count found running an active SQL statement in that interval.

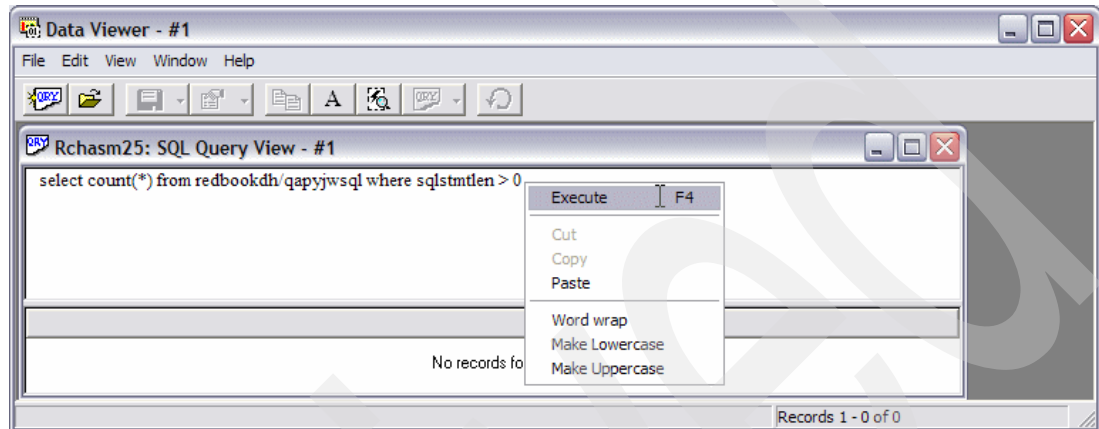


Figure 5-9 How to run an SQL statement in the full function Data Viewer

3. With the cursor positioned on the completed statement, right-click (or press F4) to run the statement. You will be prompted to select the collection member, as shown in Figure 5-10, if there are multiple collections in the QAPYJWTDE file.

We selected our collection member name **ITSO14**.

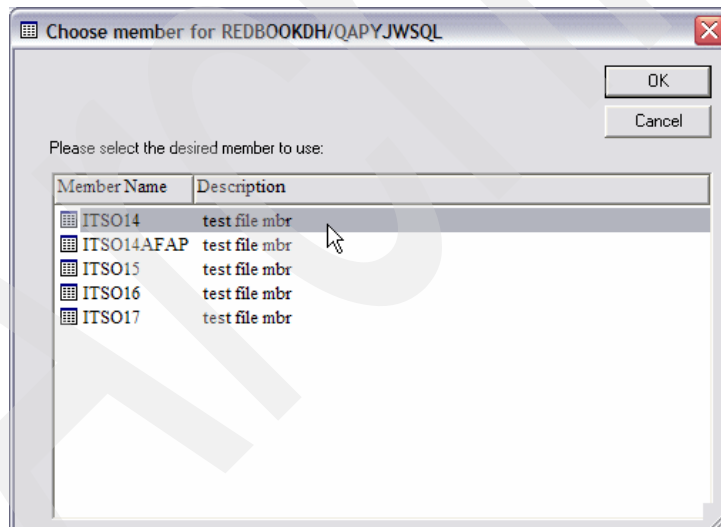


Figure 5-10 Picking a member using the full function Data Viewer

4. The query results are placed in the bottom half of the window, as shown in Figure 5-11.

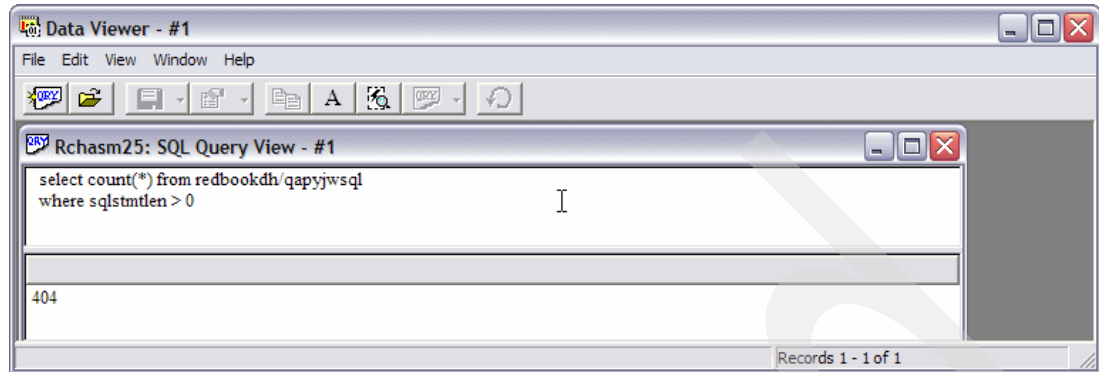


Figure 5-11 Total active SQL statement count

The total number of active SQL statements found in this collection is 404.

5. We now examine the statements, which jobs ran most SQL statements, and what user programs were running these statements.

To do this, we must run a query over data from two of the collection files:

- a. The master collection data file QAPYJWTDE

This file contains records for all task counts that either used some CPU between snapshots or are the primary task count in a multithreaded job.

This file is the *only* file that holds the job name/user/number or LIC task name for a task count.

- b. The SQL statement collection data file QAPYJWSQL

This file contains the SQL statements found active for every interval and task count in the collection. There can be one or no active SQL statements for a task count in an interval.

These SQL statements do not have the host variables and prepare information resolved; that capability is only provided in the GUI client Interval Details SQL tab.

6. Run the query shown in Figure 5-12 to find which jobs were running the most active SQL statements. We explain this first query in case you are not familiar with SQL, as its syntax is the base for several other complex queries used in this book.

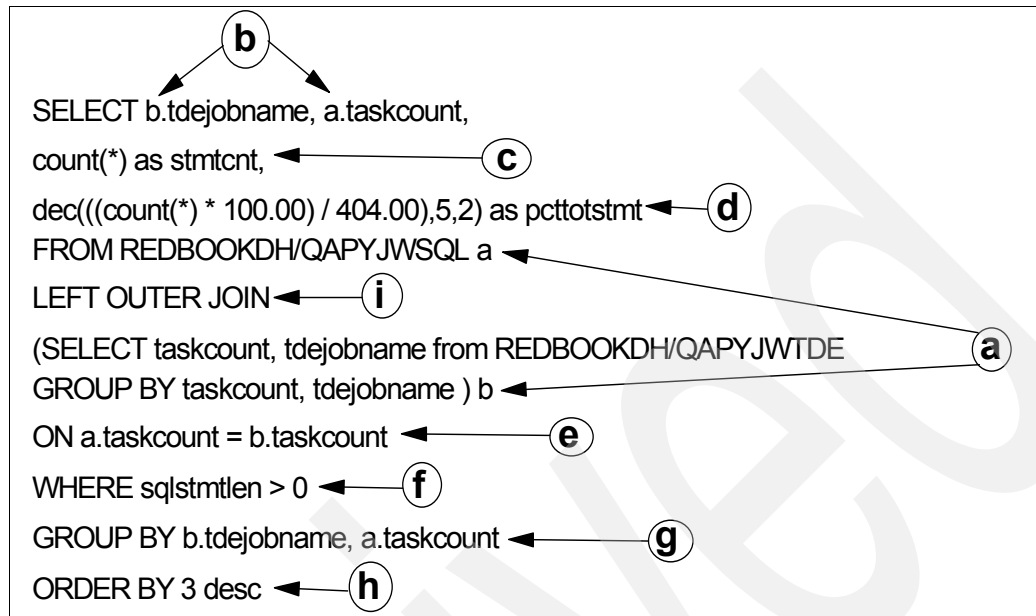


Figure 5-12 Explained SQL statement

The lower-case letters used in the following list are associated with the corresponding circled letters in Figure 5-12:

- File (SQL table) QAPYJWSQL has reference identifier of a. Where the same field name exists in more than one file, such as the TASKCOUNT field in both the QAPYJWSQL file and the QAPYJWTDE file, then using a letter prefixing the field name denotes which file the field to be used is from. In this example, the a.task count states that the task count field is to be taken from the QAPYJWSQL file.
- The TDEJOBNAME field is taken from the QAPYJWTDE file, as the SELECT statement for fields from that file is denoted by b by the brackets around the entire SELECT statement.
- The record count is assigned the field name of stmtcnt in this query.
- The percentage of the total statement count is assigned the field name of pcttotstmt in the query. This “calculation” must use the SQL *cast* as decimal function (format nnn.nn, where n is a digit 0 through 9) to get a percent value shown. If cast is not used, the result will be zero.
- The join criteria between data sets represented by a and b.
- The record selection criteria applied to the QAPYJWSQL file.
- GROUP specifies the control field breaks for summarization.
- ORDER BY specifies that the sort by field is the third (3) field out of b.tdejobname, a.task count, stmtcnt; that is, the query results are sorted by the field stmtcnt in *descending* sequence.

Note that in SQL Syntax field stmtcnt is derived—not actually in the file or table. For a derived field that has been named, instead of specifying ORDER BY 3 we could have specified ORDER BY stmtcnt.

The derived field clause shown at circle c, count(*) as stmtcnt, could have been specified simply as count(*). If that had been used, then coding ORDER BY 3 would have been required.

7. The type of join (LEFT OUTER JOIN) is used to construct the output record from fields we want to look at, from the QAPYJWSQL file and fields from the QAPYJWTDE file. LEFT OUTER JOIN is used to ensure that every record in the QAPYJWSQL file is included in the result set irrespective of whether there was a corresponding record in the QAPYJWTDE file. The QAPYJWTDE file must be used to get the job name for the task count, as that is the only file that stores the job name. The result of this query is shown in the lower pane of Figure 5-13.

Data Viewer - #1 - [Rchasm25: SQL Query View - #1]

File Edit View Window Help

SELECT b.tdejobname, a.taskcount, count(*) as stmtcnt,
dec(((count(*) * 100.00) / 404.00),5,2) as pcttotstmt
FROM REDBOOKDH/QAPYJWSQL a
LEFT OUTER JOIN
(SELECT taskcount, tdejobname from REDBOOKDH/QAPYJWTDE
GROUP BY taskcount, tdejobname) b
ON a.taskcount = b.taskcount
WHERE sqlstmtlen > 0
GROUP BY b.tdejobname, a.taskcount
ORDER BY 3 desc

Job/task name	Task count	stmtcnt	pcttotstmt
UPDADDR003DEBHAT	007054	1309	33.66
UPDADDR001DEBHAT	007052	1307	32.92
UPDADDR002DEBHAT	007053	1308	31.93
QZDASOINITUSER	007041	1245	1.23
QZDASOINITUSER	007048	1252	.24

Records 1 - 5 of 5

Figure 5-13 The task counts running the most active SQL statements

Notice that jobs UPDADDR003, UPDADDR002, and UPDADDR001 are running the most active SQL statements.

8. The next step is either to use the GUI to see the SQL statement being run for each of the identified jobs for each interval (see Figure 5-6 on page 132) or to run a query for the most popular SQL statements. We chose the query method.

To find the most popular (most frequently run) active SQL statements, we create a file containing the total active SQL statement count for our collection; this count was hardcoded as 404 in the previous examples. This file is named TOTSTMT CNT and is stored in the REDBOOKDH library. We deliberately did not place this file in the QTEMP library to ensure that it is accessible to queries run from the iDoctor Data Viewer, which runs QZDASOINIT jobs and hence would have a different QTEMP library.

Creating a total SQL statements work file for later use

We created the TOTSTMT CNT file using the i5/OS 5250 command interface command Start SQL Interactive Session (STRSQL). We could have used other iSeries SQL interfaces but not the Job Watcher SQL Data Viewer interface. The Data Viewer interface supports only the SQL SELECT statement. Here are the steps we used to create the TOTSTMT CNT file and SQL table to be used in this topic:

1. Override the Job Watcher SQL statement file to the collection member, using the Override Database File command:

```
OVRDBF FILE(APYJWSQL) TOFILE(REDBOOKDH/APYJWSQL) MBR(ITS014) OVRSCOPE(*JOB)
```

2. Create TOTSTMTCNT file/table in library REDBOOKDHT. Enter the STRSQL command, taking parameter defaults.
3. Within this interactive SQL session, enter the following SQL CREATE TABLE statement:


```
CREATE TABLE REDBOOKDH/TOTSTMT as (SELECT COUNT(*) AS TOTSTMTCNT FROM
REDBOOKDH/QAPYJWSQL where SQLSTMTLEN > 0) with DATA
```
4. Press F3 (or take the Save and Exit Session option) to end the interactive SQL session.
5. You are returned to the standard 5250 workstation interactive session. Delete the previously issued Override Database file condition:


```
DLTOVER FILE(QAPYJWSQL) LVL(*JOB)
```

We used the query shown in Figure 5-14. Remember that these counts represent the number of sample intervals in which the same SQL statement was found running. Note that without investigating the call stack, we cannot determine whether the exact same SQL statement was run over n intervals or whether the same SQL was run in each of n different intervals.

We ran all our queries from the full-function Data Viewer SQL Query view, as they each involve more than one collection file. See Appendix C, “Querying and graphing tips for Job Watcher” on page 221 to find and start the full-function Data Viewer SQL view interface.

Important: Always qualify every file name with the library name to ensure that the Data Viewer SQL Query view will prompt you for the correct member name in multiple member files (such as multiple Job Watcher collections in the same library).

a.stmtcnt	pcttotstmt	SQL statement
398	98.51	UPDATE TESTBASEPF SET F01 = F01 + 1, F02 = F02 + 1, F03 = F03 + 1, F04 = F04 + 1, F05 = F05 + 1, F06 = F06 + 1, F07 = F07 + 1, F08 = F08 + 1
1	.24	SELECT CONCAT(CONCAT(CONCAT('I, TRIM(CHAR(A.INTERVAL))), ' '), SUBSTR(CHAR(A.ISTARTTOD), 12, 12)) AS TIMEINT, ISTARTTOD
1	.24	SELECT * FROM QIDRWCHQAIDRSQ04 WHERE (SOSVRM = ? OR TRIM(SOSVRM) = ?) AND (SIDVRM = ? OR TRIM(SIDVRM) = ?) AND (SID
1	.24	SELECT * FROM REDBOOKDH/QAPYJWSQL WHERE TASKCOUNT = ? AND INTERVAL =ST
1	.24	SELECT * FROM (SELECT COUNT(*) AS WAITJOBS FROM (select COUNT(*) AS WAITJOBS FROM Redbookdh/QAPYJWDE WHERE INTERVA
1	.24	select count(*) FROM (SELECT CASE WHEN MAX(TDETYPE) <> 'T' THEN CONCAT(TRIM(SUBSTR(MAX(TDEJOBNAME), 1, 10)), CONCAT(' /
1	.24	SELECT * FROM REDBOOKDH/QAPYJWSQL WHERE TASKCOUNT = ? AND INTERVAL =,

Figure 5-14 Most popular active SQL statements

We see that the UPDATE TESTBASEPF statement was run significantly more (98.51 percent of SQL statements) than other statements. Now we want to see what jobs ran the most frequently run active SQL statements.

Therefore, we ran the query shown in Figure 5-15 on page 140 to see which jobs were running the same SQL in most sample intervals. Without more detailed investigation of the call stack, however, we cannot easily determine whether a single task count ran the same query n times or whether that task count was found running the query once for a duration of n intervals during the collection.

After viewing the results shown in Figure 5-14 on page 139 and Figure 5-15, we show you the call stack analysis as described following these figures in 5.3, “Call stack analysis for task counts” on page 140.

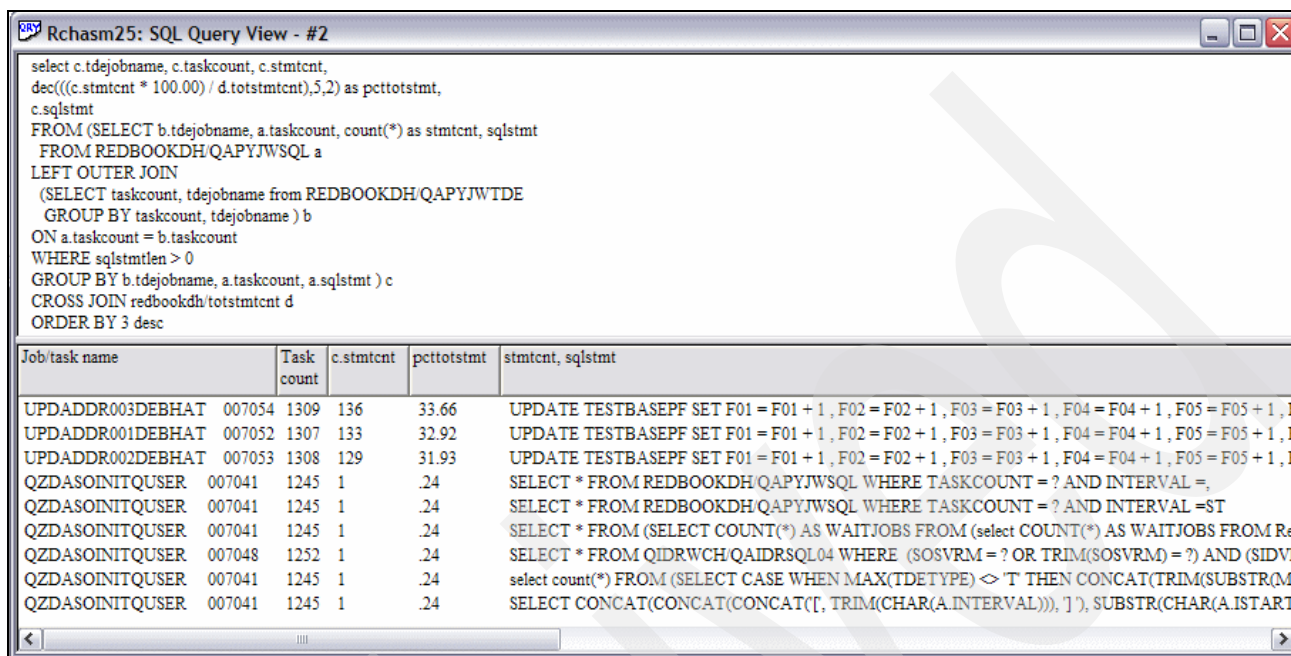


Figure 5-15 Most popular active SQL statements by task count

Note: Through V5R3 there is an iSeries SQL limit of 2000 characters on the size of the GROUP BY statement. In our example, if the variable length field sqlstmt is longer than 2000 characters, the query shown will fail.

Work file cleanup

Now that we have finished with the TOTSTMTCNT file/table created in “Creating a total SQL statements work file for later use” on page 138, we delete TOTSTMTCNT by entering an SQL Interactive Session (STRSQL) command and deleting the file:

```
DROP TABLE REDBOOKDH/TOTSTMTCNT
```

Now that you know how to find which jobs are running the most active SQL statements, you can concentrate on the top 10 or 20 of them and use the DB Monitor or the iSeries Navigator SQL Performance Monitors to capture and analyze their SQL statements for performance-tuning recommendations.

5.3 Call stack analysis for task counts

It is often necessary to find which jobs were calling program X or calling procedure Y. This can be done easily by running a query over the Job Watcher call stack data, assuming that call stack data was in the collection. This section provides an example of how to query the call stack for all task counts in the collection or for only those task counts running active SQL statements.

You should expect to modify these queries to best fit your own applications and system environment.

5.3.1 Produce the resolved call stack file

The Job Watcher collection, when selected through the wizard options, collects call stack information in an internal format within the QAPYJWSTK file. This call stack information is accessed by the Job Watcher GUI when you select the Call Stack tab when viewing, for example, Interval Details for a specific job.

In the call stack analysis examples in this chapter, we want to access the call stack information and use it to join with data from another Job Watcher file to see some interesting information. We use SQL to gather and view this information.

Because the call stack information within the QAPYJWSTK file is stored in an internal format, we have to use the Job Watcher Retrieve Call Stack Data (RTVSTKDTA) command to convert this internally formatted data to field or column data that can be accessed easily using SQL statements. Running the RTVSTKDTA command produces the converted results (exported) as a member in Job Watcher file QAIDRJWSTK.

QAIDRJWSTK has one record per call stack level per task count per interval with the associated program, module, and procedure name.

See “tRTVSTKDTA parameters” on page 250 for details of the RTVSTKDTA command parameters. We used the following RTVSTKDTA command to generate the QIDRJWSTK file (if it did not already exist) and the collection name member, ITS014, used in our example queries throughout the remainder of this chapter:

```
QIDRGUI/RTVSTKDTA WCHMBR(ITS014) WCHLIB(REDBOOKDH) MODTYPE(JW53) RCOUNT(*ALL)
TCOUNT(*ALL) STKMBRNAME(ITS014)
```

Example 5-1 shows how to join the call stack file QAIDRJWSTK to the SQL QAPYJWSQL file to get the SQL statement if one was run, and then join to the QAPYJWTD file to get the job-thread or LIC task name. Remember, you must have created a file (QAIDRJWSTK) and a specific collection member (ITS014, in our example) before running the SQL statement shown in Example 5-1.

5.3.2 The call stack for jobs with active SQL statements

The GUI shows the call stack for one task count for one interval, assuming that the call stack was collected. To see the call stack for all or some subset of task counts together with their active SQL statement, run the query shown in Example 5-1, ensuring that you have first created the QAIDRJWSTK file/member.

Example 5-1 SQL to show call stack for jobs running active SQL statements

```
SELECT  a.interval, c.tdejobname, a.frame#, a.numframes,
        a.fpgm, a.fmod, a.fproctype, a.fprocname,
        b.sqlstmt, b.sqlstmtlen, a.taskcount
FROM    REDBOOKDH/QAIDRJWSTK a
JOIN    (SELECT interval, taskcount, sqlstmt, sqlstmtlen
        FROM REDBOOKDH/QAPYJWSQL
        WHERE sqlstmtlen > 0) b
        ON a.interval = b.interval and a.taskcount = b.taskcount
LEFT OUTER JOIN
        (SELECT taskcount, tdejobname FROM REDBOOKDH/QAPYJWTD
        GROUP BY taskcount, tdejobname) C
        ON a.taskcount = c.taskcount
ORDER BY a.interval, a.taskcount, a.frame#
```

Figure 5-16 shows an example of the output from running this query from the Data Viewer.

Interval number	Job/task name	Call Stack Frame Number	Number of stack frames	Program	Module	Procedure Type	Procedure Name	SQL statement
15	UPDADDR01DEBHAT	007052	1	30		LIC	qutde_stackless_block	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	2	30		LIC	***No Procedure Found*** >	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	3	30		LIC	initializeLogicalPage_8IxRa>	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	4	30		LIC	seizeLogicalPage_8IxRadix4>	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	5	30		LIC	processPagePointerNode_8>	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	6	30		LIC	findEqualsOutline_8IxRadi>	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	7	30		LIC	insertOp_8IxRadix4FRC12>	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	8	30		LIC	insert_8IxRadix4FRC12Ix>	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	9	30		LIC	ixcbroute	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	10	30		LIC	#dbixchn	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	11	30		LIC	#dbupdim	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	12	30		LIC	#dbmodim	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	13	30		LIC	#dbupden	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	14	30		LIC	#cfmir	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	15	30		LIC	syscall_A_portal	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	16	30	QDBUDR	OPM		UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	17	30		LIC	cblabbranch	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	18	30		LIC	aiuser_program_call_portal	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	19	30	QSQRUN3	QSQUUPDAT	SQL_Update	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	20	30	QSQRROUTE	QSQRROUTE	SQL_Update	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	21	30		LIC	cblabbranch	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	22	30		LIC	aiuser_program_call_portal	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	23	30	UPDADDR5	UPDADDR5	UPDADDR5	UPDATE TESTBA
15	UPDADDR01DEBHAT	007052	24	30	UPDADDR5	UPDADDR5	_QRNP_PEP_UPDADDR5	UPDATE TESTBA

Figure 5-16 Output from query to show call stack for Jobs running active SQL statements

The Number of stack frames column indicates how many call stack levels there were for the task count when sampled. There is one record for each level in the call stack; the call stack level is shown in the Call Stack Frame Number column. The lower the value in this Call Stack Frame Number column, the more recent the program/module/procedure call occurred. A call stack entry (frame) number of 1 indicates that the associated program/module/procedure was the most recent call.

In this example we see that:

- ▶ The first interval (15) in which job UPDADDR01, user DEBHAT, number 007075, task count 1307 (not displayed in our example, but is in the query as column a.taskcount) was running UPDATE TESTBASEPF
- ▶ The most recent user program on the call stack was program UPDADDR5 at stack level 23
- ▶ Program UPDADDR5 was running SQL as it called QSQRROUTE as shown at stack level 20. Program UPDADDR5 is an ILE program, shown by the Procedure Type column, and it is ILE RPG as shown by the call to the ILE PEP for RPG of _QRNP_PEP_ under the procedure column.
- ▶ The QSQRROUTE SQL router program called the SQL Update procedure at stack level 19.
- ▶ The SQL Update procedure calls the i5/OS program QDBUDR, which represents database update/delete/release.
- ▶ The active SQL statement being run at this time is UPDATE.

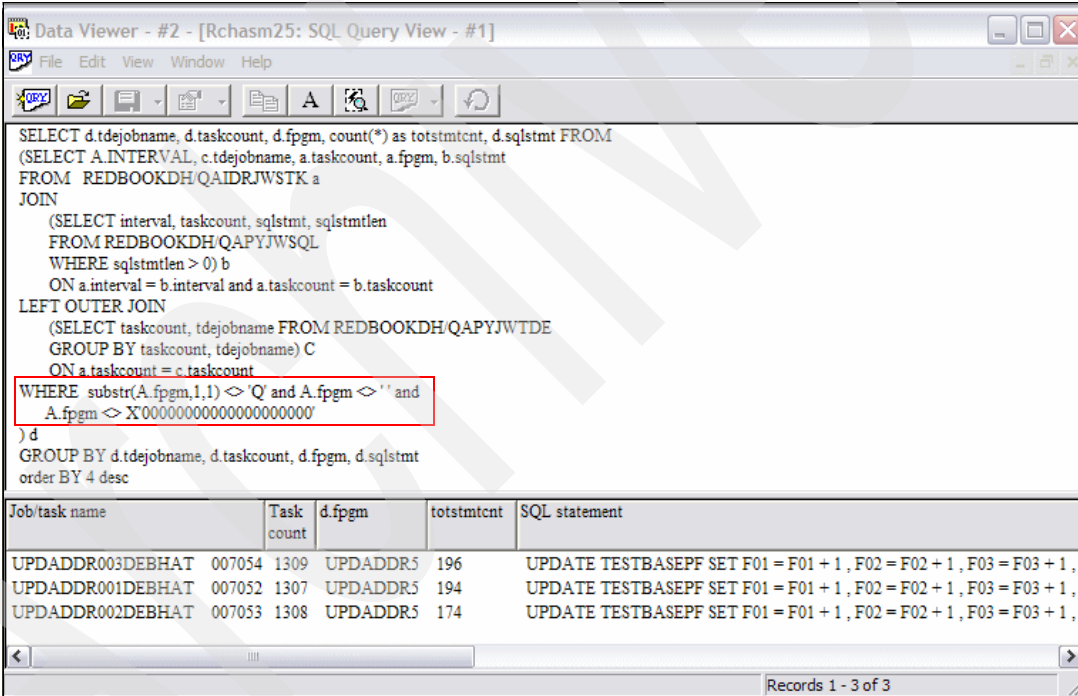
This chapter is focused on SQL applications, so in the following topics we show refinements to our basic call stack query example. Our example query can be changed to select or not select records or rows with certain i5/OS program and LIC microcode task-naming syntax.

Our examples should be used as suggestions for the kinds of queries you can write to drill down into more detailed Job Watcher and SQL-specific information to help in your performance problem determination process.

Our examples follow in the next section, 5.3.3, “Finding user programs running active SQL statements” on page 143, and 5.3.4, “Finding any SQL-related or database-related system activity” on page 143.

5.3.3 Finding user programs running active SQL statements

The query shown in Example 5-1 on page 141 must be modified to exclude all i5/OS system programs (beginning with the letter Q), and exclude all SLIC task procedures to show only user program names. Since the QAIDRJWSTK file can also have either blanks or X'00000000000000000000' stored as a program or task name, both of these values must be excluded from the query, in addition to any programs beginning with Q.



The screenshot shows a 'Data Viewer' window with a SQL query and its results. The query is as follows:

```
SELECT d.tdejobname, d.taskcount, d.fpgm, count(*) as totstmtcnt, d.sqlstmt FROM
(SELECT A.INTERVAL, c.tdejobname, a.taskcount, a.fpgm, b.sqlstmt
FROM REDBOOKDH/QAIDRJWSTK a
JOIN
(SELECT interval, taskcount, sqlstmt, sqlstmtlen
FROM REDBOOKDH/QAPYJWSQL
WHERE sqlstmtlen > 0) b
ON a.interval = b.interval and a.taskcount = b.taskcount
LEFT OUTER JOIN
(SELECT taskcount, tdejobname FROM REDBOOKDH/QAPYJWSTDE
GROUP BY taskcount, tdejobname) C
ON a.taskcount = c.taskcount
WHERE substr(A.fpgm,1,1) <> 'Q' and A.fpgm <> '' and
A.fpgm <> X'00000000000000000000'
) d
GROUP BY d.tdejobname, d.taskcount, d.fpgm, d.sqlstmt
order BY 4 desc
```

The results table has the following data:

Job/task name	Task count	d.fpgm	totstmtcnt	SQL statement
UPDADDR003DEBHAT 007054	1309	UPDADDR5	196	UPDATE TESTBASEPF SET F01 = F01 + 1 , F02 = F02 + 1 , F03 = F03 + 1 ,
UPDADDR001DEBHAT 007052	1307	UPDADDR5	194	UPDATE TESTBASEPF SET F01 = F01 + 1 , F02 = F02 + 1 , F03 = F03 + 1 ,
UPDADDR002DEBHAT 007053	1308	UPDADDR5	174	UPDATE TESTBASEPF SET F01 = F01 + 1 , F02 = F02 + 1 , F03 = F03 + 1 ,

The status bar at the bottom indicates 'Records 1 - 3 of 3'.

Figure 5-17 User programs running most active SQL by job

In this example we can see that the user program UPDADDR5 is running all of the active SQL statements in the collection.

5.3.4 Finding any SQL-related or database-related system activity

In this topic we show how to subset call stack data to find where system SQL-related work or system database routines are being run. The query shown in Example 5-18 on page 144 finds where system database routines (programs or procedures beginning with QDB in the name) are being run or where system SQL-related work (programs or procedures beginning with QSQ in the name).

Data Viewer - #1 - [Rchasm25: SQL Query View - #1]								
File Edit View Window Help								
SELECT a.interval, c.tdejobname, a.frame#, a.numframes, a.fpgm, a.fmod, a.fproctype, a.fprocname, a.taskcount FROM REDBOOKDH/QAIDRJWSTK a LEFT OUTER JOIN (SELECT taskcount, tdejobname FROM REDBOOKDH/QAPYJWTD GROUP BY taskcount, tdejobname) C ON a.taskcount = c.taskcount WHERE (substr(a.fpgm,1,3) = 'QDB' or substr(a.fmod,1,3) = 'QDB' or substr(a.fprocname,1,3) = 'DB' or substr(a.fprocname,1,3) = 'qdb' or substr(a.fpgm, 1,3) = 'QSQ' or substr(a.fmod,1,3) = 'QSQ' or substr(a.fprocname,1,3) = 'QSQ' or substr(a.fprocname,1,3) = 'qsq') ORDER BY a.interval, a.taskcount, a.frame#								
Interval number	Job/task name	Call Stack Frame Number	Number of stack frames	Program	Module	Procedure Type	Procedure Name	Task count
14	QDBFSTCCOLQSYS 006858	14	17	QDBSTSRV	QDBSTSRV	ILE	_CXX_PEP_Fv	671
15	UPDADDR001DEBHAT 00>	16	30	QDBUDR		OPM		1307
15	UPDADDR001DEBHAT 00>	19	30	QSQRUN3	QSQUPDAT	ILE	SQL_Update	1307
15	UPDADDR001DEBHAT 00>	20	30	QSQRUN3	QSQRUN3	ILE	SQL_Update	1307
15	UPDADDR002DEBHAT 00>	16	30	QDBUDR		OPM		1308
15	UPDADDR002DEBHAT 00>	19	30	QSQRUN3	QSQUPDAT	ILE	SQL_Update	1308
15	UPDADDR002DEBHAT 00>	20	30	QSQRUN3	QSQRUN3	ILE	SQL_Update	1308
15	UPDADDR003DEBHAT 00>	18	32	QDBUDR		OPM		1309
15	UPDADDR003DEBHAT 00>	21	32	QSQRUN3	QSQUPDAT	ILE	SQL_Update	1309
15	UPDADDR003DEBHAT 00>	22	32	QSQRUN3	QSQRUN3	ILE	SQL_Update	1309
16	QDBSRVXR QSYS 006852	6	12	QDBXREF		OPM		665
16	QDBSRVXR QSYS 006852	9	12	QDBXREFM	QDBXREFM	ILE	QDBXREFM	665
16	QDBSRVXR2 QSYS 006856	6	12	QDBXREF2		OPM		669
16	QDBSRVXR2 QSYS 006856	9	12	QDBXREFM	QDBXREFM	ILE	QDBXREFM	669
16	UPDADDR001DEBHAT 00>	16	30	QDBUDR		OPM		1307

Figure 5-18 Finding jobs running system-SQL or system database program procedures

Here you can see the last entry job/task count for collection interval 14 is for an i5/OS IBM database server job which does some table-column cross-reference processing and the first entries for interval 15, which are for our UPDADDR001/02/03 jobs (with the job number component truncated (00>)). You can see that updates are being performed by these jobs (program names QDBUDR (read for update), QSQRUN3 (router to specific SQL function programs), and SQL update (QSQUPDAT)).

Observing job QDBSRVXR, we can deduce the updating is probably to at least some index (SQL key) field (SQL column).

5.4 Journal analysis

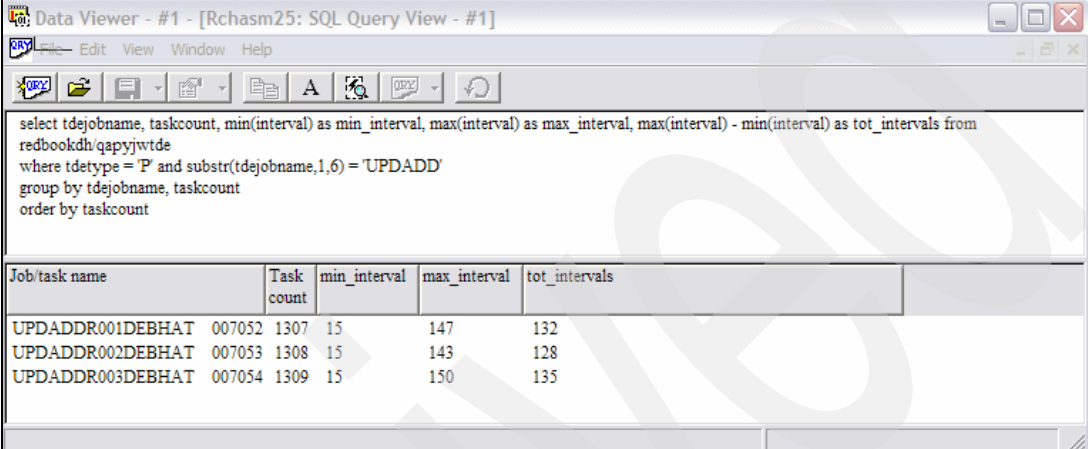
If your environment is doing a significant amount of database activity that is being journaled, you should assess the impact of the journaling itself on performance. We installed the high availability journal performance feature #42 for i5/OS (journal caching) on our system in order to measure the effects of running a journaling application both without and then with this journal-caching feature. There is no commitment control in our application.

This section shows how Job Watcher can be used to show bottlenecks and the effect of changes on performance. The first thing we need to do is to find out when the first and last collected intervals are in the QAYPETDE file for the jobs in the example application.

5.4.1 No journal caching

Normally you would not have the journaling cache feature installed when you do your analysis. In our test environment, however, we had it installed. We turned off the system's ability to use the caching feature by specifying STRJRN JRNCACHE(*NO).

The first thing we needed to do was to determine when our jobs started and ended. We used the Data Viewer to query the collection master file, QAPYJWTDE, to find the first and last intervals in which our jobs did any work, as shown in Figure 5-19.



Job/task name	Task count	min_interval	max_interval	tot_intervals
UPDADDR001DEBHAT	007052 1307	15	147	132
UPDADDR002DEBHAT	007053 1308	15	143	128
UPDADDR003DEBHAT	007054 1309	15	150	135

Figure 5-19 No journal caching: minimum and maximum intervals for workload

In this example, our selection criteria (where clause) selects Program TDE types with the first six characters of the job name being UPDADD. We also see that the first interval in which our workload ran was 15 and the last interval was 150.

The valid tdtype field/column values are defined in this book in Appendix B, “Database files created by Job Watcher” on page 199; for the QAPYJWTDE file at “The master file QAPYJWTDE for jobs, threads, and tasks” on page 208.

We changed the Job Watcher graphing of vertical bars to be 138 wide instead of the default of 50 so that we can see the entire workload in the Collection Overview Signature, as shown in Figure 5-20.

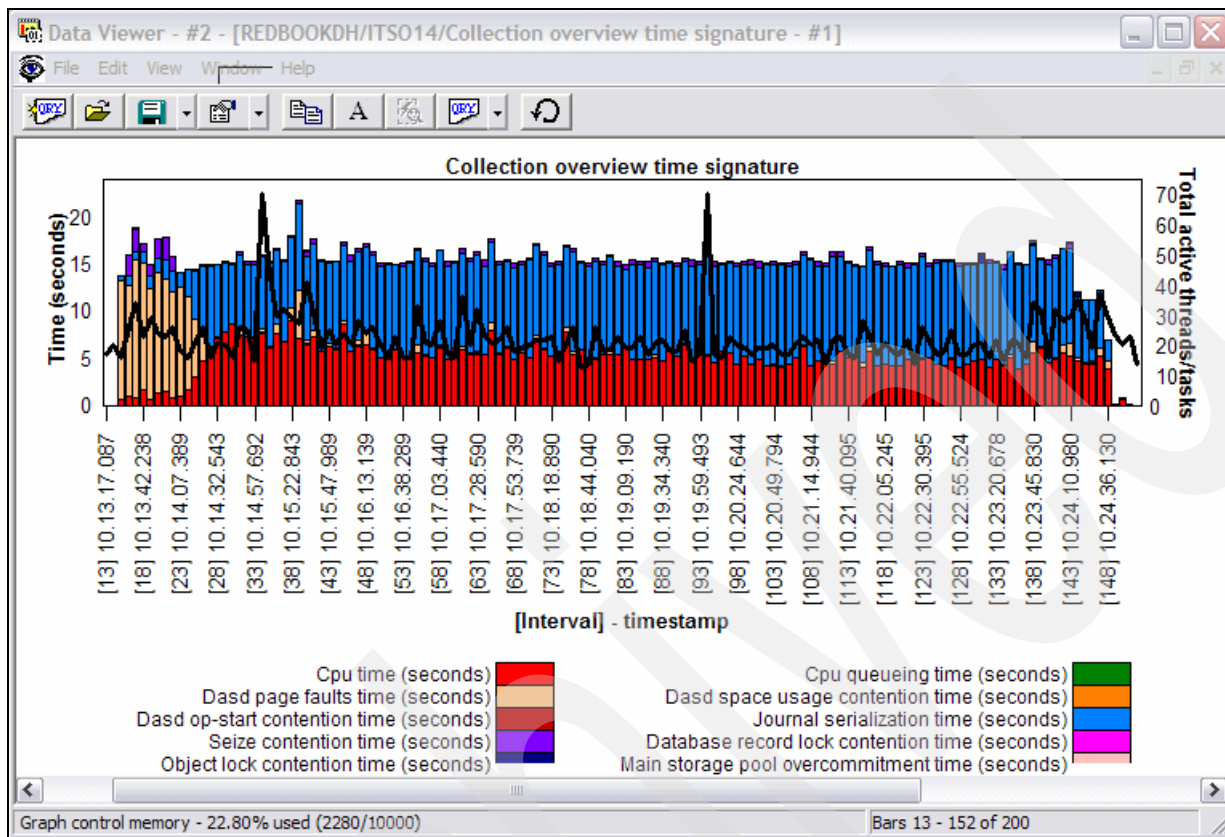


Figure 5-20 No journal caching: Collection overview time signature for the whole workload

As we briefly addressed in the text describing Figure 5-4 on page 130, starting at interval 15 there are a lot of page faults that require investigation.

Page faults

Figure 5-20 shows a burst of page faulting; therefore, we must identify what objects are being heavily affected and, if possible, what was actually issuing the page faults.

Right-click in interval 15 and select the **DASD Page Faults** → **Faults Graph by Job** graph, which gives us the results shown in Figure 5-21 on page 147.

In this figure we see that for interval 15, the three main jobs issuing (causing) page faults were application jobs UPDADDR002, UPDADDR001, and UPDADDR003.

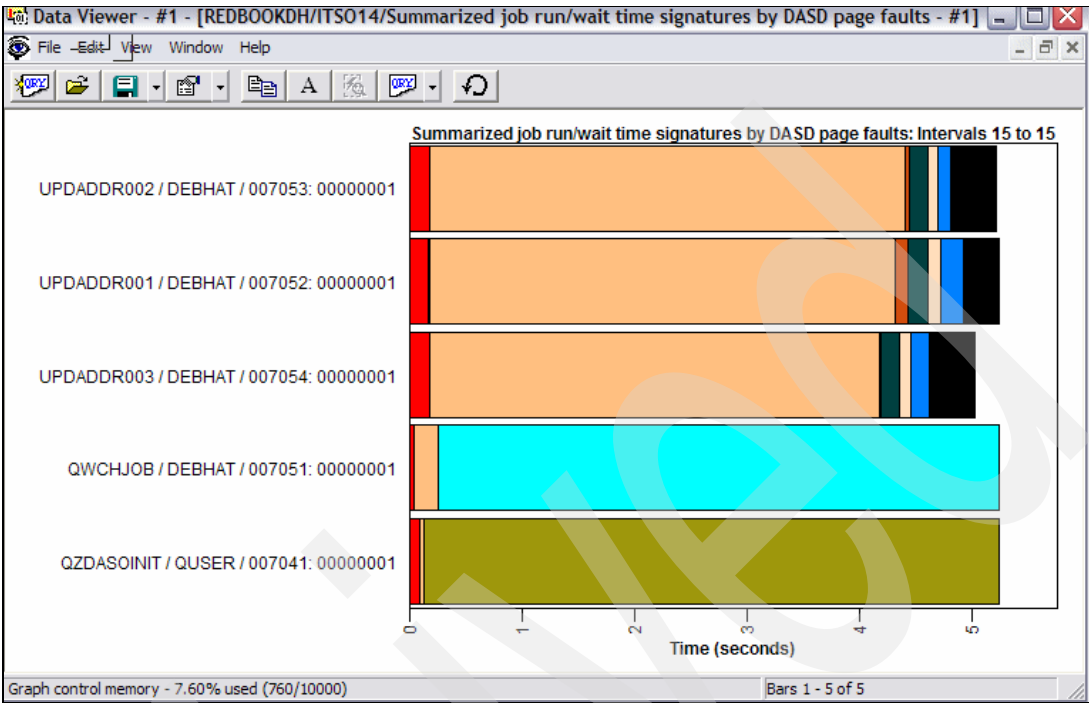


Figure 5-21 DASD page faults by job for interval 15

What objects were these jobs faulting against?

Right-click the first job, UPDADDR0002, to select **Detail Records**. This shows all waits for the job in interval 15. We only want the waits on Page Faults, so we refined our query to select only records where the Current or Last Wait Bucket (field BLOCKBKT in file QAPYJWTDE) is 5 (DASD Page Faults). Before capturing this window, we rearranged the fields to clearly show the objects being faulted against. The results are shown in Figure 5-22.

REDBOOKDH/ITSO14/Detail records for UPDADDR002 / DEBHAT / 007053: 00000001 - #2								
Interval number	Task count	Current or last blocking bucket	Total time in current wait in microsecs	Wait object name		Wait object obj type description	Wait object type in hex	Current or last LIC wait object
15	1308	5	6690	TESTLF01	TESTLF01	DB2 ACCESS PATH	0C90	SFt
16	1308	5	7760	TESTLF04	TESTLF04	DB2 ACCESS PATH	0C90	SFt
17	1308	5	4052	TESTLF19	TESTLF19	DB2 ACCESS PATH	0C90	SFt
18	1308	5	10244	TESTLF13	TESTLF13	DB2 ACCESS PATH	0C90	SFt
19	1308	5	40622	TESTLF03	TESTLF03	DB2 ACCESS PATH	0C90	SFt
20	1308	5	14743	TESTLF05	TESTLF05	DB2 ACCESS PATH	0C90	SFt
21	1308	5	1783	TESTLF16	TESTLF16	DB2 ACCESS PATH	0C90	SFt
23	1308	5	13953	TESTLF07	TESTLF07	DB2 ACCESS PATH	0C90	SFt
24	1308	5	9137	TESTLF10	TESTLF10	DB2 ACCESS PATH	0C90	SFt
								161

Figure 5-22 DASD page faults: objects being faulted against

We see now that these faults were on DB2 access paths for objects named TESTLF01, TESTLF04, and so forth. These are either logical files or keyed (indexed) physical files. This is normal for the startup of our example application and is not a problem. Note that logical file pages are normally brought in with page faults rather than explicit reads.

Figure 5-23 on page 148 shows that the work on the system is fairly similar for each of the intervals after interval 27. We will examine interval 27 in more detail to see precisely what is

happening here. Notice that most of the time is in journal serialization wait and the rest is in CPU. (This is “dispatched for use” CPU time).

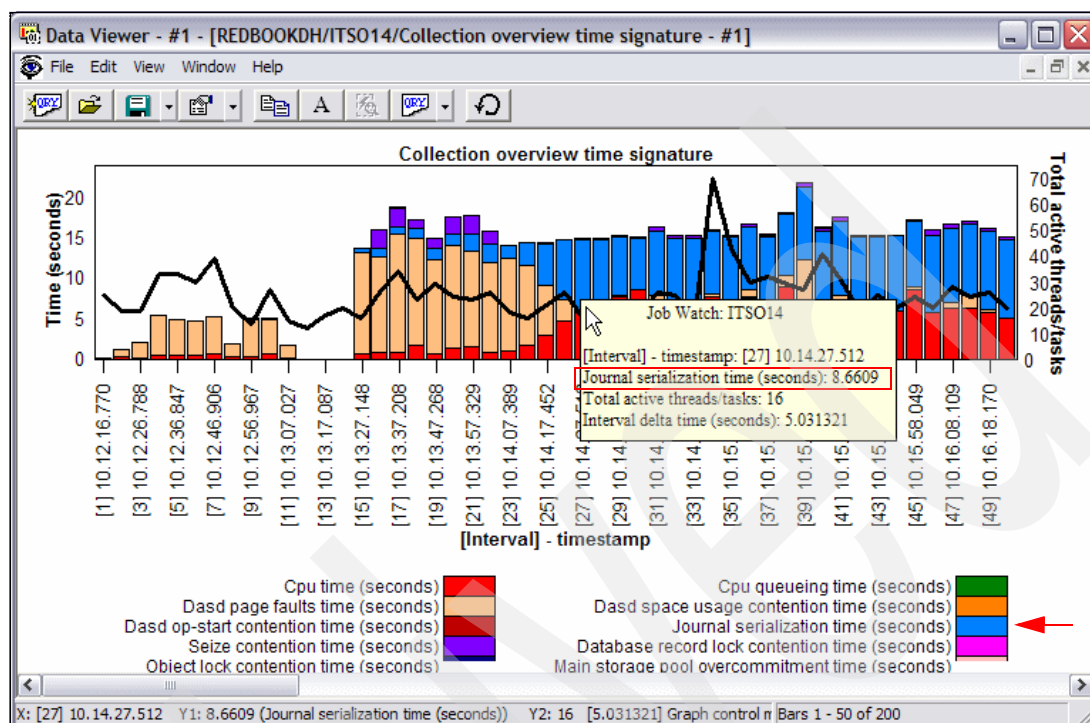


Figure 5-23 No caching: pick a typical interval for the workload

We need to know the wait profile for our application jobs during interval 27. You can see that 8.6609 total seconds of time was spent in journal serialization wait by *all active task counts* in interval 27 that were included in the definition of the Job Watch collection.

Important: As stated in 2.2, “Job Watcher terminology” on page 23, a task count is a symbolic representation of a job or task or thread that is active on the system. Task counts do not actually use CPU or do SQL or anything else: the jobs, tasks, and threads do that.

Right-click in interval 27 and select **Journal serialization** to produce a graph for each of the task counts (that are actually in this interval,\ that used more than 0 CPU seconds of time). These are sorted in journal serialization wait descending sequence, as shown in Figure 5-24.

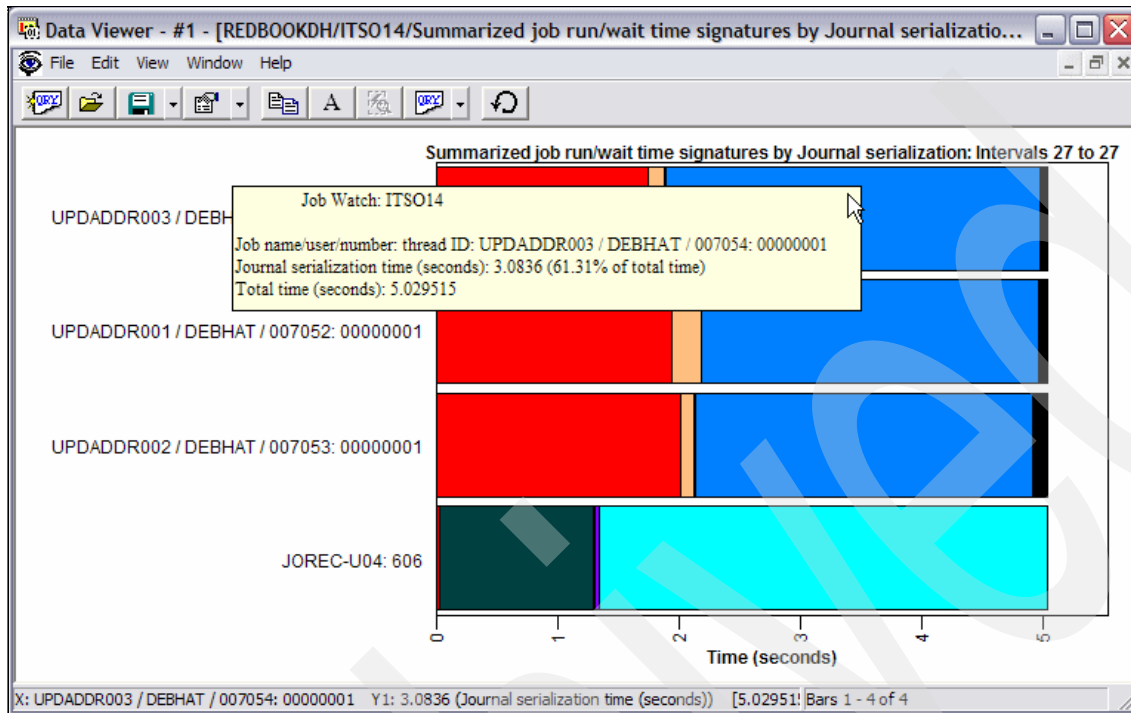


Figure 5-24 No journal caching: expanded typical interval (27) showing jobs in journal waits

All three UPDADDR* application jobs seem to be spending most of their time in journal waits or using the CPU *in this interval*, so we see whether these jobs have the same wait profile for all intervals.

Right-click and select **Wait graphs** → **Run/wait time signature** to see the collection-wide run/wait signature for these jobs. We opened them into the same Data Viewer for ease of comparison. See Figure 5-25 on page 150 for the results.

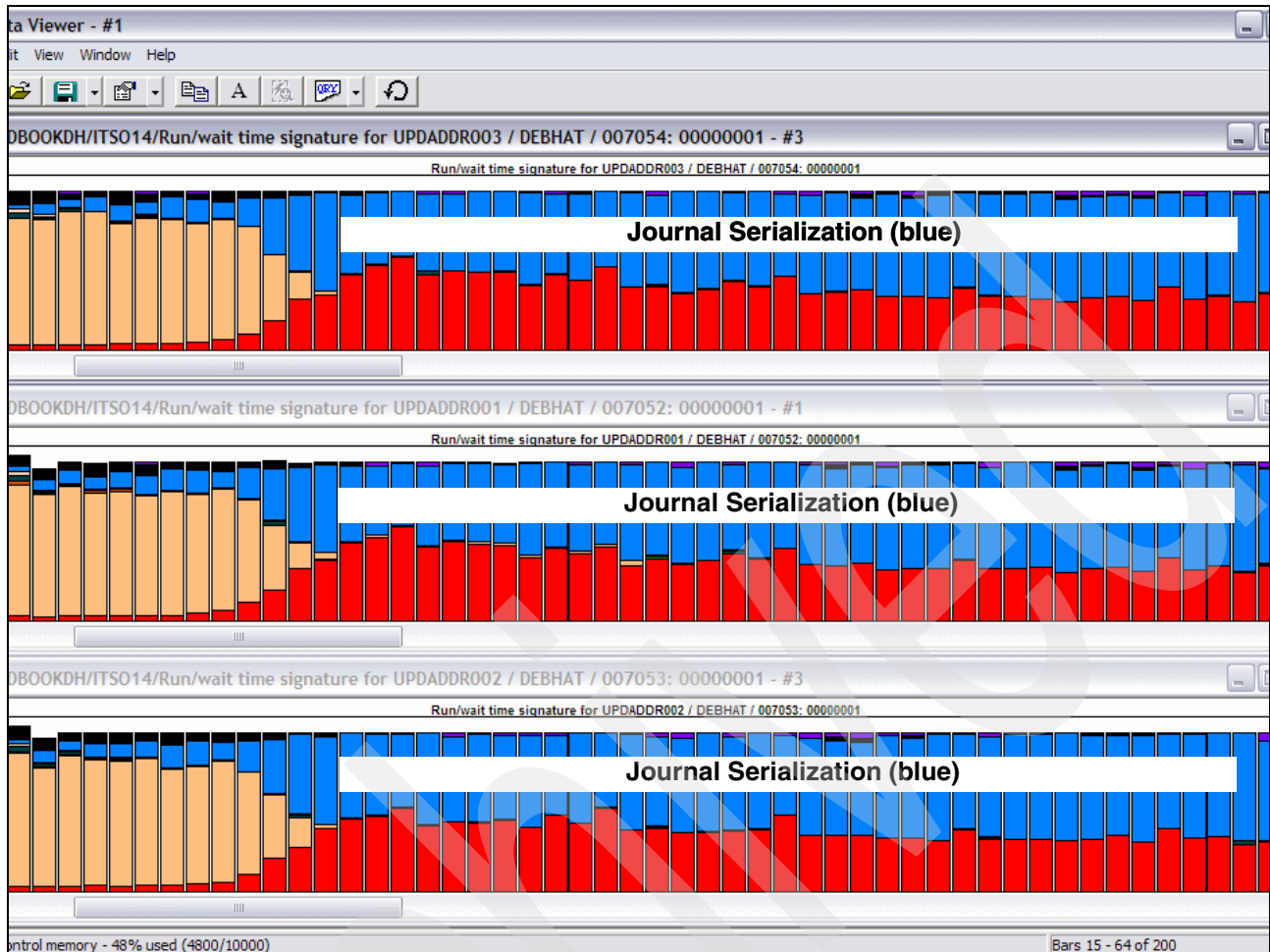


Figure 5-25 No journal caching: wait signature for the three application jobs

It is now clear that these jobs are all spending most of their time in journal wait (serialization) in all of the intervals from interval 27.

What is causing these journal serialization waits?

First, we check the interval 27 details to see what is happening by clicking inside interval 27 on the journal serialization wait area of the graph (Figure 5-23 on page 148). This produces the interval details as shown in Figure 5-26. Recall that the interval details are for all jobs and LIC tasks that used >0 CPU in that interval.

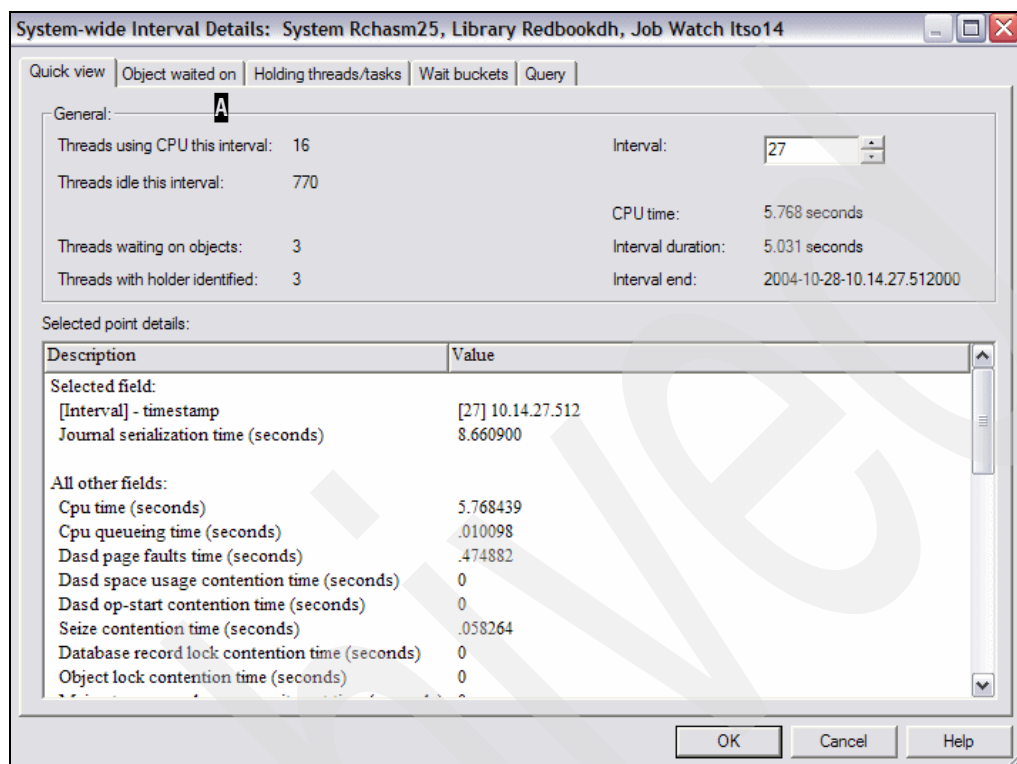


Figure 5-26 No journal caching: interval 27 details

This figure confirms that there was a total of 8.660900 seconds of journal serialization wait time for all task counts collected in interval 27. To understand what these waits were on, we click the Object waited on tab (A), to examine the objects being waited on as shown in Figure 5-27.

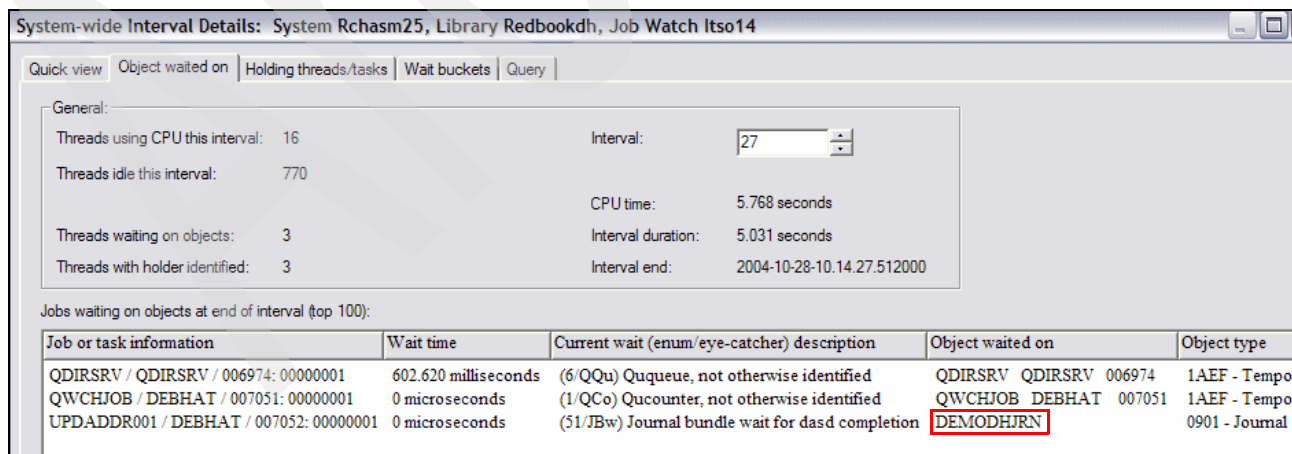


Figure 5-27 No journal caching: interval 27 detail: objects being waited on

This shows us that the journal object DEMODHJRN was involved in a journal bundle wait on DASD completion. However, at this level, it looks like the wait time was zero. We need to do

further investigation to determine whether journal serialization waits are part of a performance problem.

We next select the **Holding threads/tasks** tab to see waits during this interval. See the results in Figure 5-28.

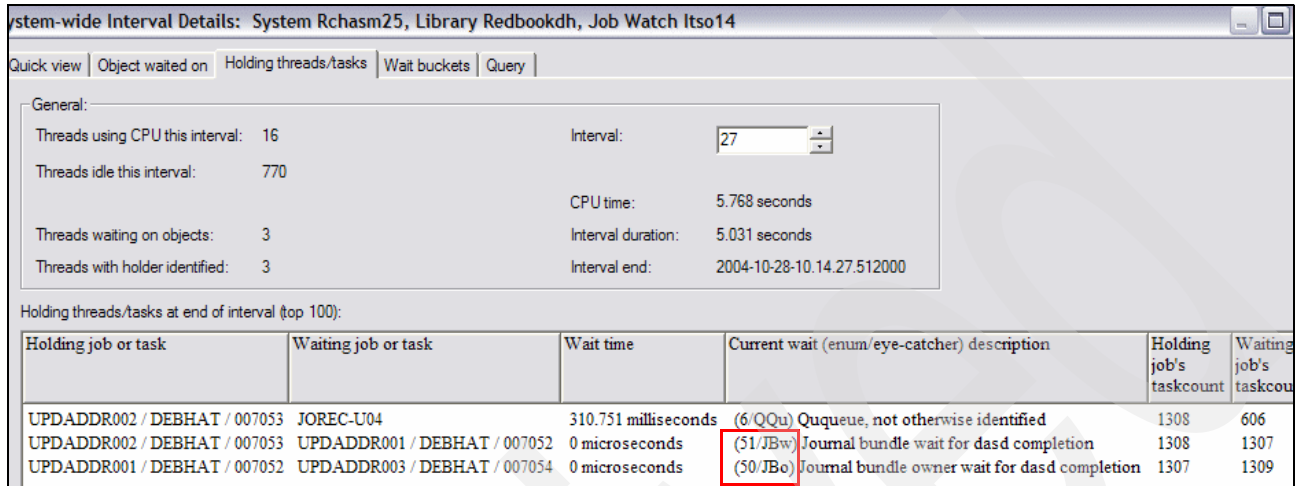


Figure 5-28 No journal cache: interval 27 detail: wait holders

Clearly there were two journal serialization waits: ENUMs 51 and 50, which map into the journal serialization wait bucket number 13. (See “Job Watcher wait points (ENUMs) and wait buckets” on page 30 for ENUM-to-wait-bucket mappings.) However, these waits are shown as 0, indicating that they have just started at the time the snapshot was taken. Next, we select the **Wait buckets** tab to view the wait buckets information for this interval. See Figure 5-29.

Note: Since we originally clicked in the journal Wait time area, we are only seeing all the wait buckets where the journal wait time was > 0. This is not all of the waits occurring in this interval. To see all the waits for the interval, change **Sort by** to **All Waits**.

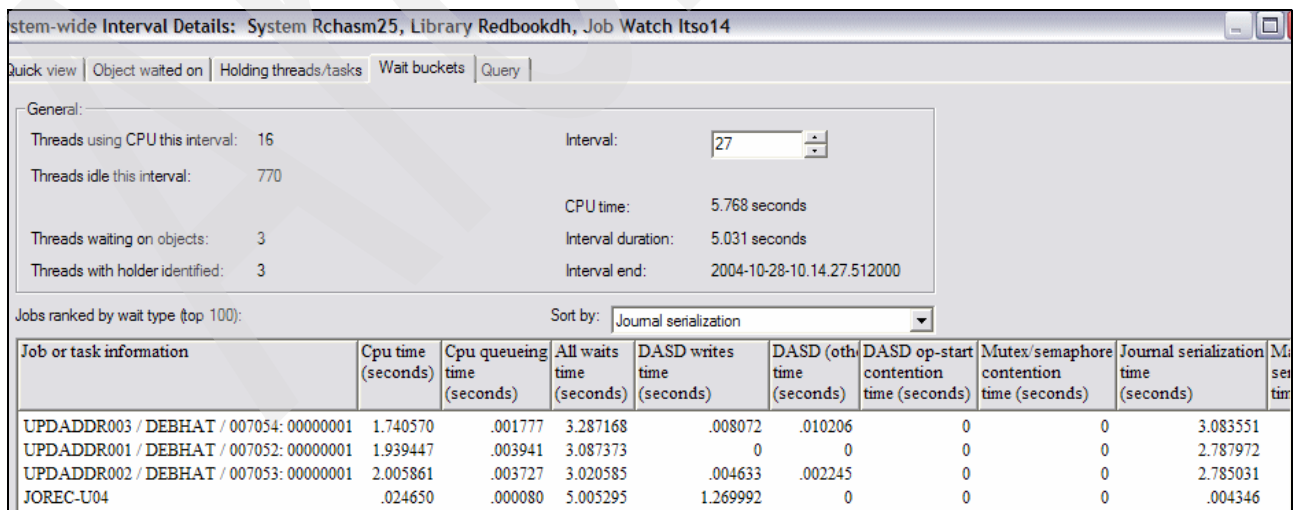


Figure 5-29 No journal cache: interval 27 detail: wait buckets

Look at all of the column values, specifically the All waits time (seconds) and Journal serialization time (seconds) columns. This emphasizes that the journal serialization waits are the majority of the wait time for this interval. Why did we see them as 0 previously? This is because we used a 5-second snapshot. This reminds us again that Job Watcher is a *sampling tool* and not a tracing tool; we are not catching many of these waits in progress.

How can we see this? Query the QAPYJWTDE file to see both the total time and number of these journal waits; the query used is shown in Example 5-2 and the results are shown in Figure 5-30.

Example 5-2 SQL to produce Figure 5-30

```
SELECT taskcount, tdejobname, sum(qcount13) as totcount13,
      sum(tdeusecs) as inttotus, sum(qtime13) as tottime13,
      CAST((double(sum(qtime13)) / Double(sum(tdeusecs)) * 100) AS DECIMAL(12, 8)) as PctJrnwait,
      sum(qtime13) / sum(qcount13) as avgjrnwait,
      sum(deltacpu) as totdeletcpu, sum(qtime01) as tottime01
FROM redbookdh/qapyjwtde
WHERE taskcount = 1308 or taskcount = 1309 or taskcount = 1307
GROUP BY taskcount, tdejobname
ORDER BY avgjrnwait asc
```

Task count	Job/task name		totcount13	inttotus	tottime13	PctJrnwait	avgjrnwait	totdeletcpu	tottime01
1308	UPDADDR002DEBHAT	007053	84135	648921484	386220221	59.51724985	4590	154536775	202725783
1309	UPDADDR003DEBHAT	007054	84204	684045494	392783369	57.42065000	4664	156547131	220373206
1307	UPDADDR001DEBHAT	007052	83810	668957623	392170979	58.62418866	4679	154330561	217364035

Figure 5-30 No journal caching: evaluating journal serialization details

Using the wait bucket description in “Bucket 13: Journal serialization” on page 40 and the QAPYJWTDE file field descriptions in “The master file QAPYJWTDE for jobs, threads, and tasks” on page 208, we describe the derived fields shown in the columns of the query results.

- **totcount13:** This is the total number of all journal serialization waits over the collection. QAPYJWTDE qcount13 is the journal serialization count for each collection interval. qcount13 is correlated with qtime13 used in this SQL example.
- **tottime13:** This is the total microseconds of journal serialization wait time over the collection. QAPYJWTDE qtime13 is the journal serialization wait time within each collection interval.
- **inttotus:** This is the total microseconds of all intervals of the collection. QAPYJWTDE tdeusecs is the delta time between Job Watcher intervals.
- **PctJrnwait:** This is the derived percentage of the total of microseconds of all intervals in the collection that were spent in journal serialization wait. In simple terms this is the percent of the entire collection time that had journal serialization waits going on.

For the three jobs (tasks, threads) shown, each was waiting for journal entries to be made in the range of 57% to 59% of total collection time. This means that for each of our three jobs, an average of more than 2.5 seconds of every 5-second interval was spent waiting, with an average of 0.004600 seconds (avgjrnwait column) per journal serialization wait.

These are significant values and naturally lead to the question of whether the additional cost i5/OS journal caching feature would greatly improve this situation.

Note that we had to use the SQL CAST as DECIMAL function to get the percentage to be computed properly. Without using this function the percentage value was shown as 0. Since the QAPYJWTDE file fields used in the calculation have 0 decimal positions, and the result is a fraction (digits to the right of the decimal point), standard SQL syntax truncates the quotient results to zero.

- ▶ **avgjrnwait:** This is the computed average wait time for each journal serialization wait by dividing the total journal serialization wait time by the number of journal serialization waits.
- ▶ **totdeltcpu:** This is the total number of microseconds each job (thread, task) used during the entire collection. QAPYJWTDE deltcpu is the CPU time in microseconds used since the last interval snapshot.
- ▶ **tottime01:** This is the total in microseconds that the job was dispatched to a CPU during the entire collection. APYJWTDE qtime01 is the CPU time the job was dispatched to a CPU during an interval snapshot.

Within a system or partition that is extremely overloaded at the CPU resource level, the dispatched CPU value could be longer than actual CPU used. In most environments the dispatched CPU value and the actual CPU used value are nearly identical.

If we reran the collection using a much smaller sample interval (0.1 second or using “as fast as possible”) we would see that these journal serialization waits are all JBo (Journal Bundle Owner wait for DASD completion) or JBw (Journal Bundle wait for DASD completion). These are waits on the journal object for the journal entries to be written out to disk.

Tip: The totcount13, tottime13, ... PctJrnwait, ...through tottime01 columns are derived and exist only for the life of the query. The SELECT statement can be saved for later use. See “Saving your new query definition” on page 232.

You can also assign more meaningful text to the query results column heading than, for example, the derived PctJrnwait column. See “Changing field (column) headings in your query results” on page 235.

The primary ways to reduce this journal serialization wait time include:

- ▶ Get the additional cost journal caching option called High Availability Journal Performance available as feature 42 of i5/OS.
- ▶ Use commitment control.
- ▶ Order and install faster technology I/O hardware. This would include newer IOPs such as the #2844 PCI IOP, disk controllers, such as the 2757 or 2780, and faster disk drives. Customer feedback on this newer disk-related hardware features has demonstrated significant performance improvement over previous-generation hardware.

Note that you can review disk service times in the Performance Tools for iSeries reports.

In this book we concentrate on the journal caching option as it involves no application programming changes, only a simple change to the application journal startup process.

5.4.2 Journal caching

With the high availability journal caching feature installed, we started the journaling with the cache function activated, using the Start Journaling (STRJRN) command with JRNCACHE(*YES). Then we ran a Job Watcher collection with our example application running again.

We used the Data Viewer to query the collection master file, QAPYJWTDE, to again find the first and last intervals in which our jobs did any work, as shown in Figure 5-31 on page 155.

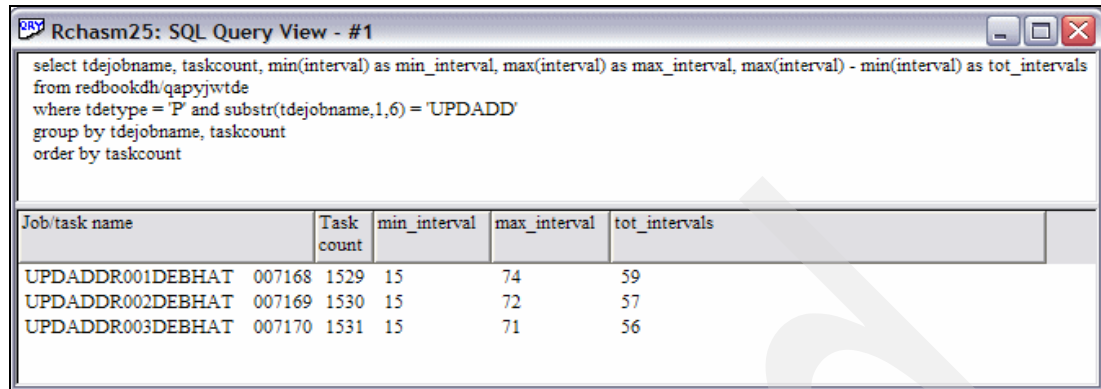


Figure 5-31 Journal caching: minimum and maximum intervals for workload

We see that the first interval in which our workload ran was 15 and the last interval was 74. We changed the Job Watcher graphing of vertical bars to be 62 wide instead of the default of 50 so that we can see the entire workload in the Collection Overview Signature, as shown in Figure 5-32.

Note: Our workload runs much faster. Now it takes only 60 intervals to run instead of 136.

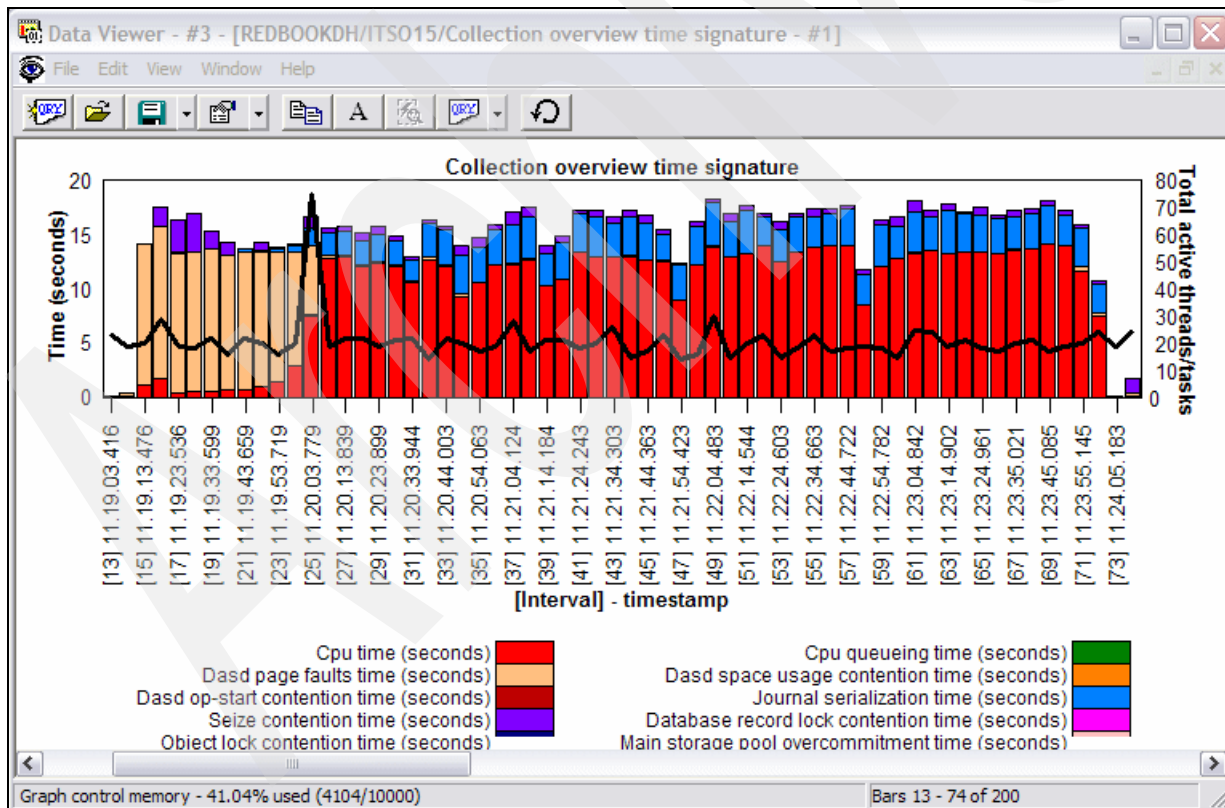


Figure 5-32 Journal caching: collection overview time signature for the whole workload

The elapsed time for the workload has decreased considerably. The work has also been compressed with more CPU time used in each interval and less journal wait time in each

interval. Opening both collections into the same Data Viewer as shown in Figure 5-33 and ensuring the same scaling on the x and y axes, we can easily see the differences.

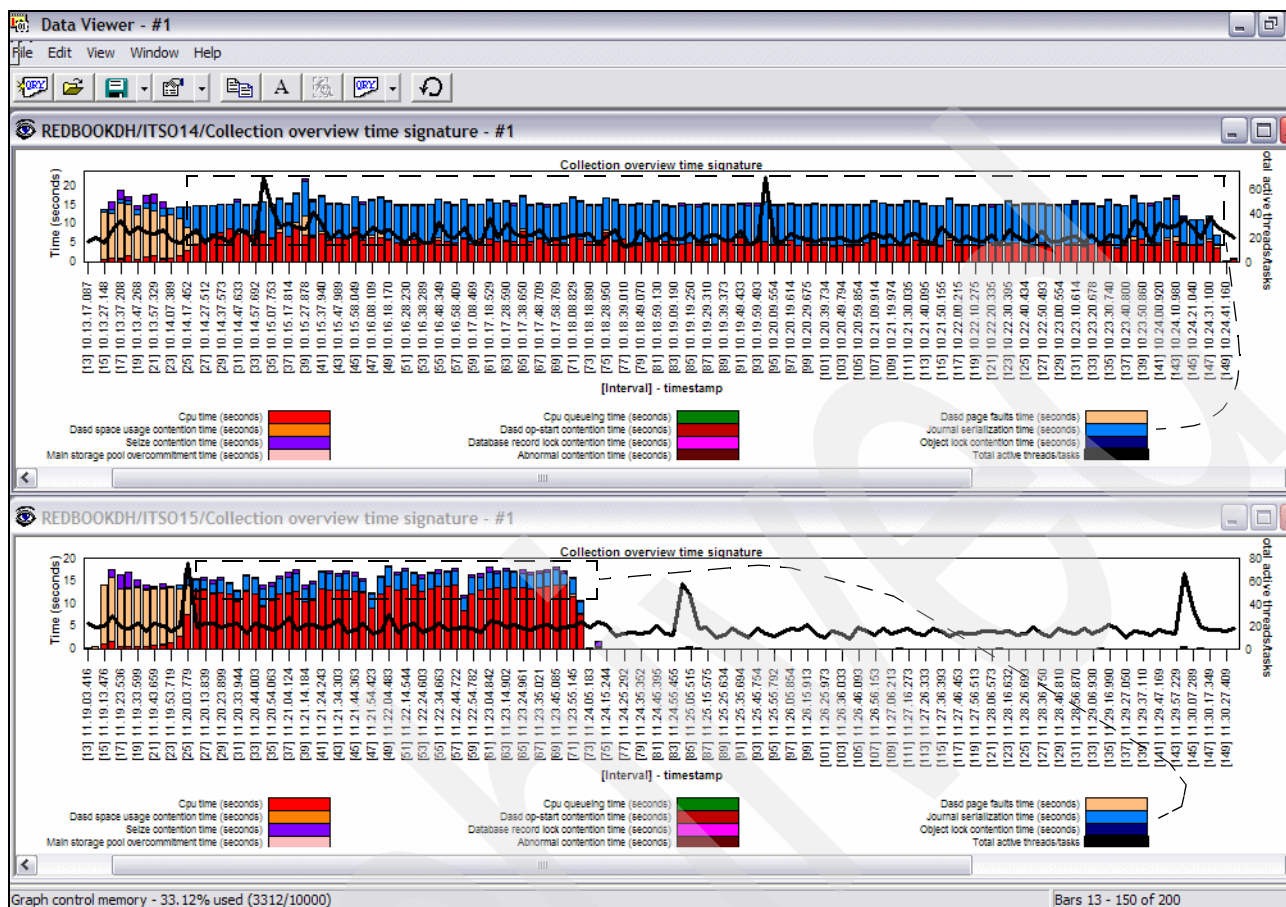


Figure 5-33 Comparing the workload with and without journal caching

In this figure we highlighted the area identified as journal serialization time with a black dashed line and square in the interest of making this wait identifiable in case you are reading this book in black-and-white printout instead of online, where the blue color representing journal serialization can be distinguished more easily from the red CPU time.

Compared to the upper figure (without journal cache), the lower graph demonstrates journal serialization wait time has been greatly reduced. And, of course, the job took fewer intervals to complete.

Figure 5-34 shows a typical interval's details for the journaling cache results.

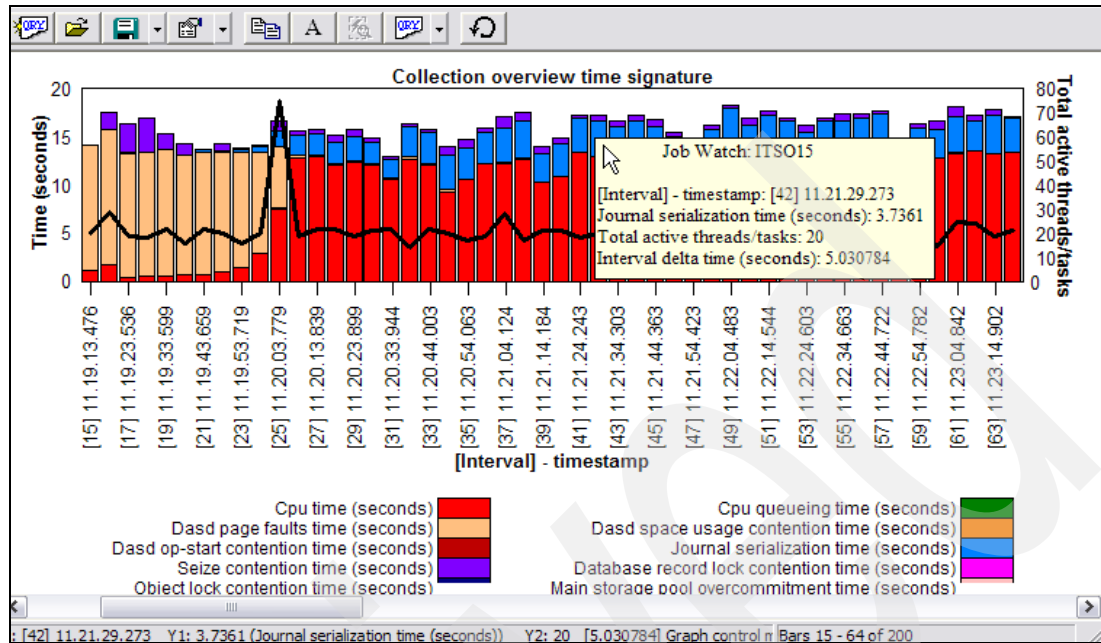


Figure 5-34 Journal Caching: Pick a typical interval for the workload

Using 42 as a typical interval (and as a reminder that it is the feature code on i5/OS for journal cache), we can see that the journal serialization time for the workload interval is now 3.7361 seconds, compared to 8.6609 seconds without the journal caching.

There is no need to drill down further.

We conclude that using Job Watcher to view our workload with and without journal caching, the use of journal caching has:

- Made our workload run faster
- Significantly reduced the journal serialization wait time
- Made our workload more CPU intensive

The test jobs are running faster and using the same amount of CPU time in less elapsed time. This causes the CPU utilization to go up and, depending on CPU intensity of other jobs running on the system or partition, could cause higher CPU queueing for other jobs.

Archived

Analysis example: Java application

This chapter discusses two very common types of problems for Java applications that often run in WebSphere® Application Server environments, and how to use Job Watcher and Heap Analyzer to collect, analyze, and make decisions to correct the situation.

There is no information in this chapter specific to running Java programs under a WebSphere Application Server environment. However, before doing Java-based performance problem analysis, we suggest you that you have already examined the performance tips documented:

- ▶ At the iSeries Performance Management Web site:
<http://www-1.ibm.com/servers/eserver/iseries/perfmgmt/webjtune.html>
- ▶ In the redbook *Maximum Performance with WebSphere Application Server V5.1 on iSeries*, SG24-6383:
<http://www.redbooks.ibm.com/abstracts/sg246383.html>

This chapter presumes that you have read the job waits descriptions in Chapter 2, “Overview of job waits and iDoctor for iSeries Job Watcher” on page 21. We also presume you are familiar with the contents of Chapter 3, “Getting started” on page 51.

If you are new to Java but must do iSeries-based performance analysis, consider reading any of several commonly available Java-overview publications that can be found in most bookstores and, of course, at the Sun Microsystems Web site:

<http://www.sun.com>

For an eServer i5 or iSeries system perspective on Java, we suggest using the iSeries Information Center Web site:

<http://www.ibm.com/eserver/iseries/infocenter>

Select your region and language for V5R3. In the left navigation bar, expand **Programming** → **Java** → **IBM Developer Kit for Java platform**. Click **Java platform** and select one of the articles.

6.1 Two problem scenarios

The two reasonably common problems associated with a Java application performance problem are:

- ▶ CPU “misusage”
- ▶ Heap growth

We use a query example of Job Watcher collected call stack information in this chapter. To understand this query and write your own queries, refer to the wait bucket definitions listed in Table 2-3 on page 30 and the Job Watcher file and field descriptions included in “Job Watcher terminology” on page 200.

6.2 CPU “misusage”

We use the term *CPU misusage* within this book to address two different CPU utilization scenarios:

- ▶ CPU utilization is higher than expected. Your response time or throughput is not what is expected.
- ▶ CPU utilization is too low. You are experiencing poor performance while the system resources are barely being used.

Java applications running outside of or under WebSphere Application Server servers are available in all levels of complexity—just like applications written in any programming language. Some applications handle very high volumes of data and drive lots of work through the system, while others use very little system resources.

You can, in theory, have the same kind of Java application performance problem in two very different application environments. For example, one is consuming 25% of the available processor power, while the other is consuming 95% of the processor power. In this CPU usage topic, we use Job Watcher to investigate these situations and our approach is the same.

When you have a Java application performance problem, typically the first thing to examine is the CPU utilization of Java-based jobs. Your WebSphere Application Server environment might be using more CPU than you expect it to. This value could be 25% of your processor or 99%; it does not matter because our problem analysis approach is the same.

Use the following commands to determine if there are too many jobs on the system or if some jobs are using a large percentage of processor time (for example, your Java jobs), in comparison to other active jobs:

- ▶ Work with Active Jobs (WRKACTJOB)
- ▶ Work with System Activity (WRKSYSACT)

These commands are covered in Chapter 1, “iDoctor for iSeries Job Watcher positioning” on page 1.

6.2.1 Job Watcher

After you determine that you have a problem to investigate, you should decide whether Job Watcher is the right approach. Job Watcher is a very powerful tool for this type of problem, but it is not a replacement for Collection Services or even a simple Trace Profile (TPROF) collection. We often use PEX TPROF to get to the job or procedure name that is causing the

problem. But if we want to investigate CPU use at a thread level within the job, then Job Watcher is very good at providing this type of information.

Job Watcher startup options

Startup Options are where you define your collection as required (Figure 6-1). When you are trying to track down CPU it is best to collect as much information as you can when you are experiencing the problem. We typically suggest checking the **Collect as fast as possible** option and set **Maximum data to collect** to 2 to 5 GB of data, assuming that you have the disk storage space available. The collection name and library for the data can be anything you choose. Click **Configure**.

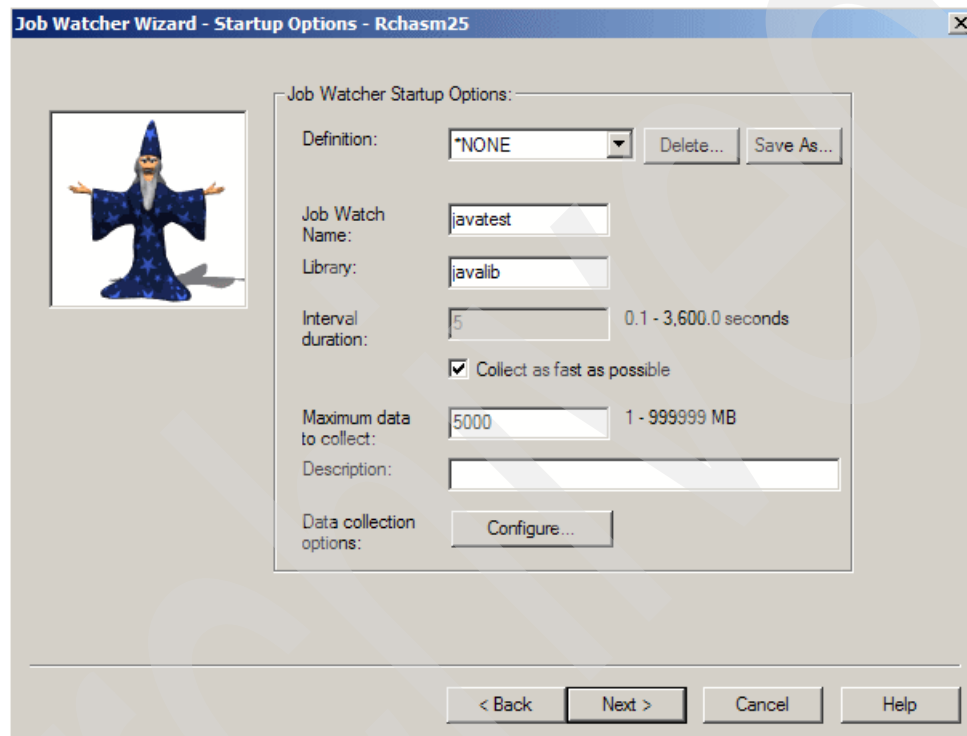


Figure 6-1 Job Watcher Wizard: Startup Options

Job Watcher Data Collection Options

Under Data Collection Options, you further define the collection. For CPU-related issues, the Call Stack is a very important piece of information. This is what enables us to understand what particular threads are doing. For this type of problem you want the stack data on every interval, as shown in Figure 6-2. Click **OK**.

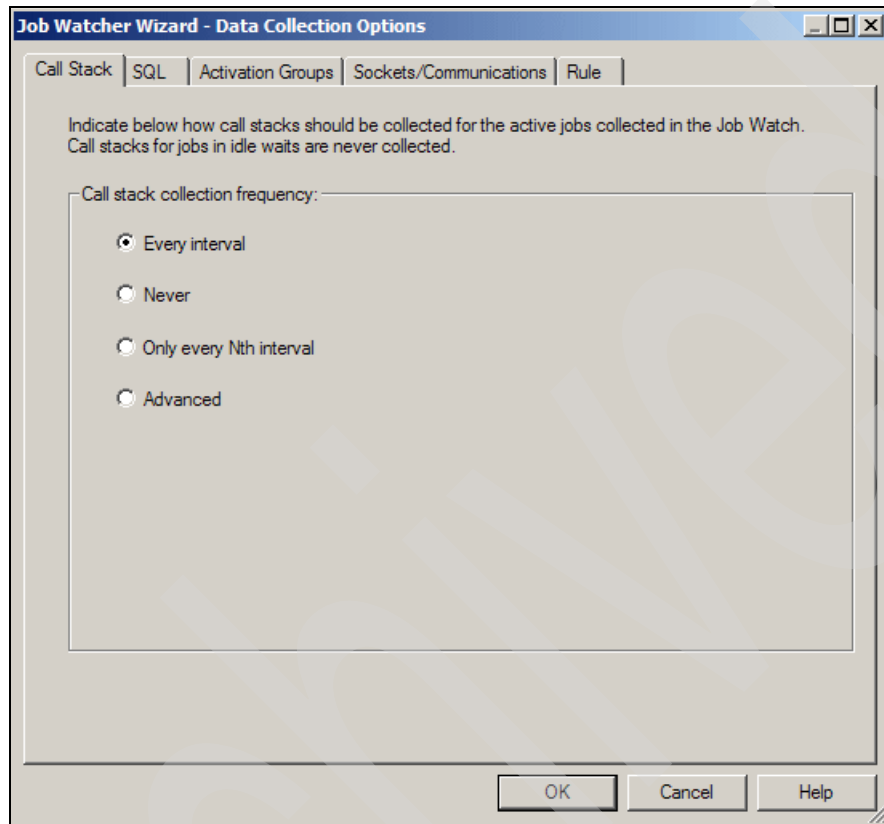


Figure 6-2 Job Watcher Wizard: Data Collection Options

Job/Task Option

You have the option to filter your collection to a specific job or group of jobs, as shown in Figure 6-3. If your environment is experiencing very high CPU utilization, your Job Watch collection has less impact if you specify as few jobs and tasks as possible. In cases similar to our example, you should already have a good idea of what jobs you are interested in, so isolate your collection to the job or jobs of interest.

1. Click **Select specific jobs and tasks**. Click **Next**.

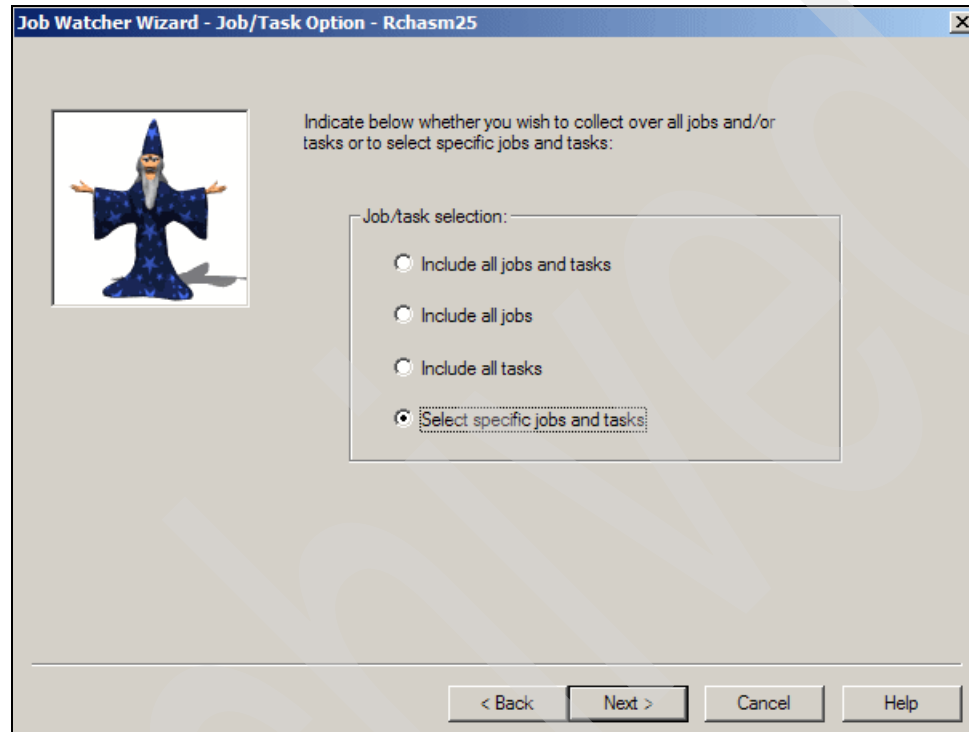


Figure 6-3 Job Watcher Wizard: Job/Task Option

2. Make sure the **Select by** parameter list specifies **Job name** as shown in Figure 6-4. Click **Add** to identify the job or groups of jobs you are looking for.

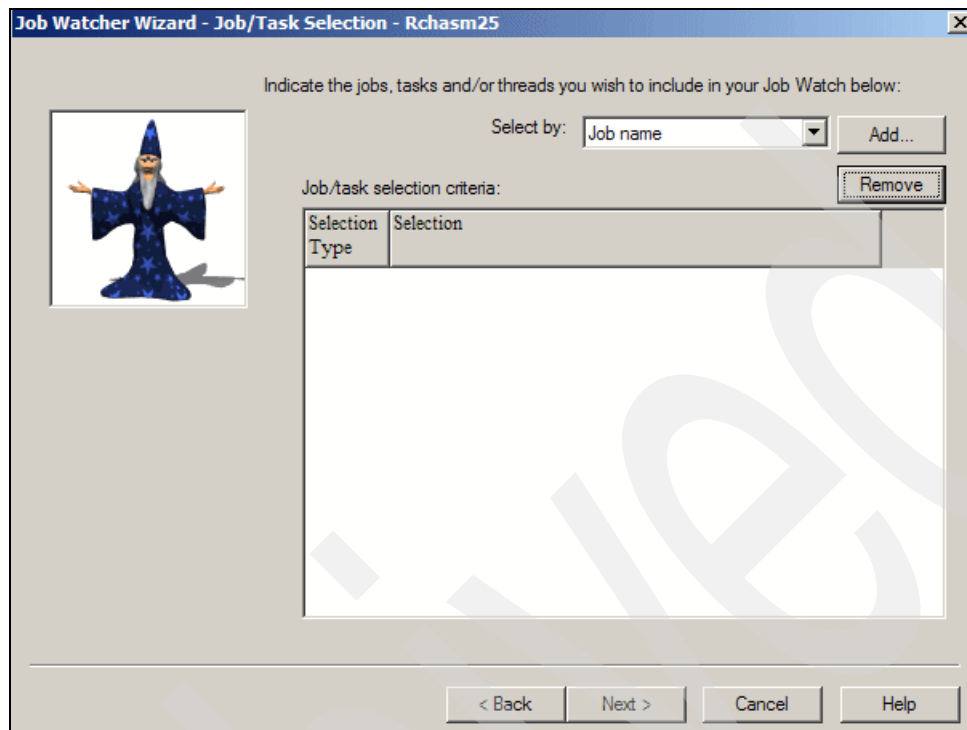


Figure 6-4 Job Watcher Wizard: Job/Task Selection

3. Enter any Job Filter Information you want and select **Refresh** for a list of available jobs. Highlight the job and click **Add Selected** as shown in Figure 6-5.

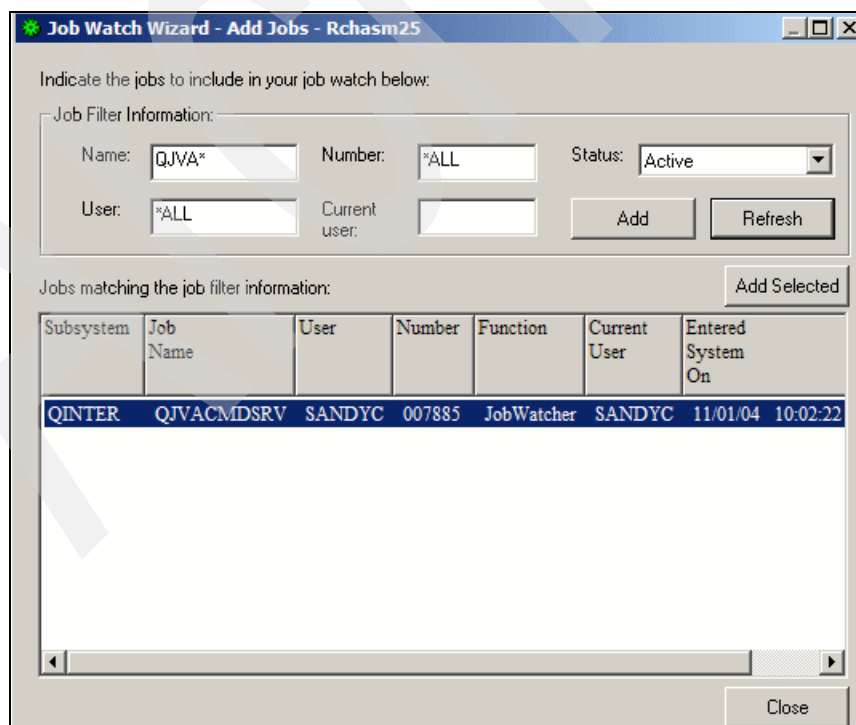


Figure 6-5 Job Watcher Wizard: Add Jobs

Ending Options

You can further limit the collection based on a number of intervals or time. As a starting point, we suggest collecting for a specific time duration and setting this to 10 minutes as shown in Figure 6-6. Click **Next**.

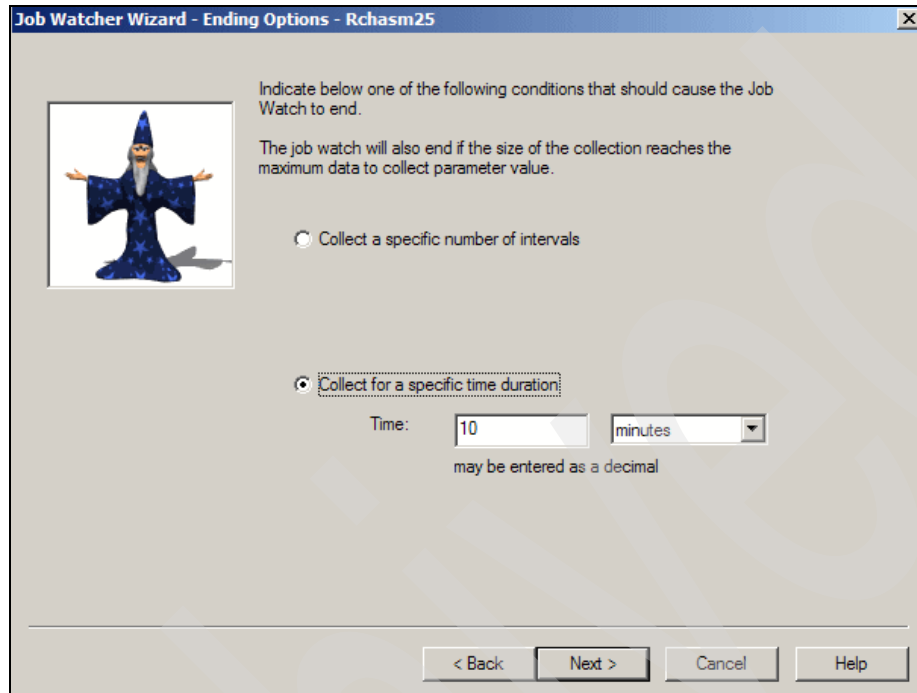


Figure 6-6 Job Watcher Wizard: Ending Options

Review the summary window as shown in Figure 6-7 and make sure you have the collection set up as you want. Then click **Finish** to submit the job.

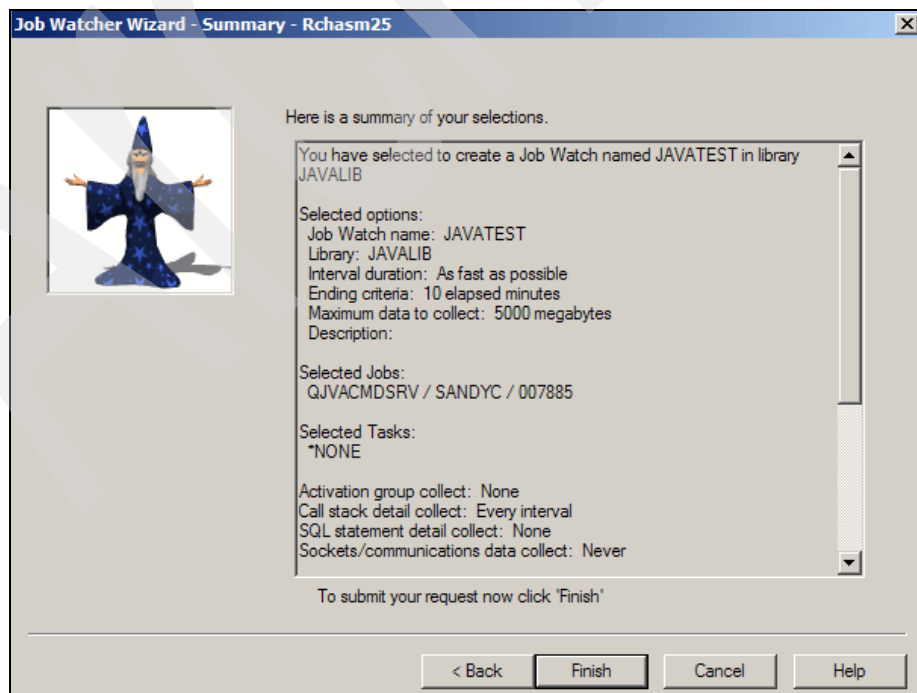


Figure 6-7 Job Watcher Wizard: Summary

Viewing the collected data

When the collection is running, refresh the library view on the left side of the Job Watcher window and select your collection library to display the list of watches that are available. Double-click the watch that you just started. In our example, we are chasing a CPU misuse problem, so we start with the CPU graphs (Figure 6-8). Click on CPU graphs. From the list of available reports in the right-side pane, right-click the report for **CPU/CPUq usage by high/low priority** to open the graph in the Data Viewer.

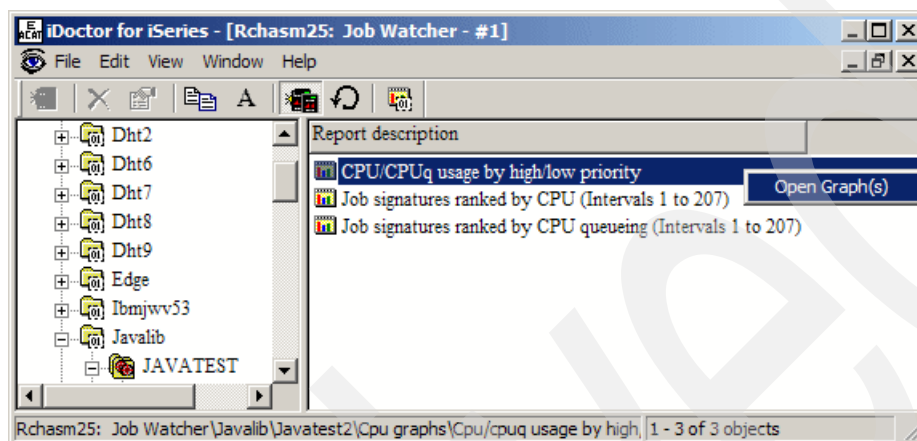


Figure 6-8 Open CPU/CPUq usage by high/low priority graphs

The graph in Figure 6-9 shows CPU activity by interval. It is very common when chasing an increase in CPU that one or two threads will stand out. For this reason it is a good idea to look at a graph of the *CPU usage by thread* to see whether there are any threads we should investigate. Choose one interval in the CPU/CPUq usage by high/low priority graph that looks interesting.

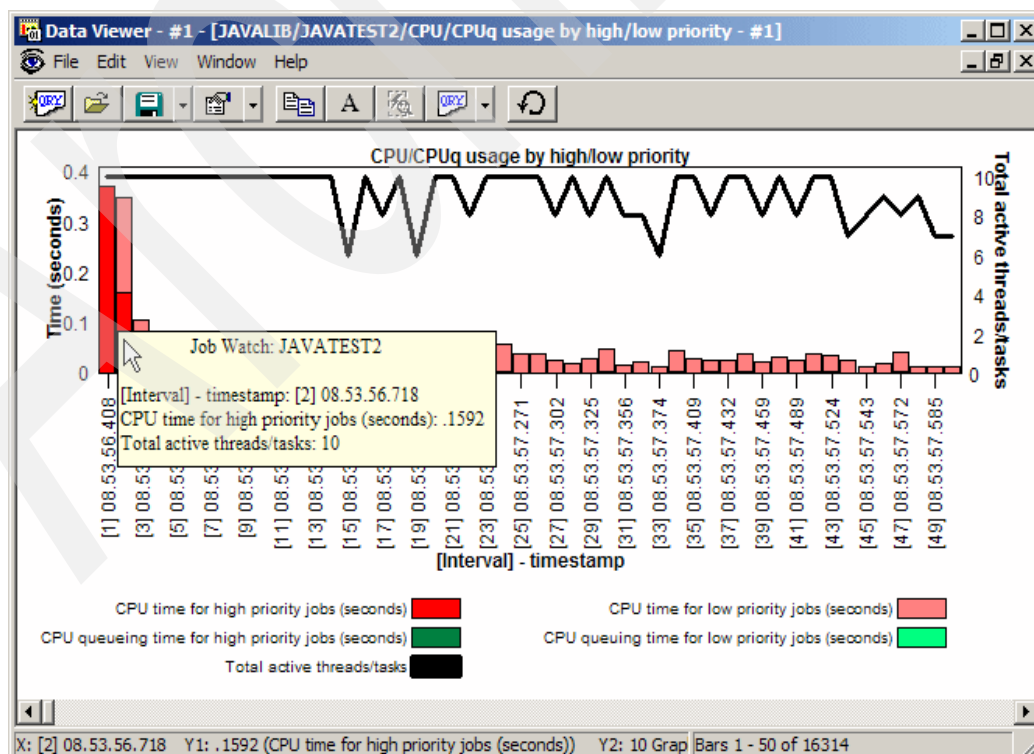


Figure 6-9 CPU/CPUq usage by high/low priority

In this example, we look at interval 2. Right-click the vertical bar in interval 2 for a list of detailed reports as shown in Figure 6-10, and select **Threads using CPU this interval**.

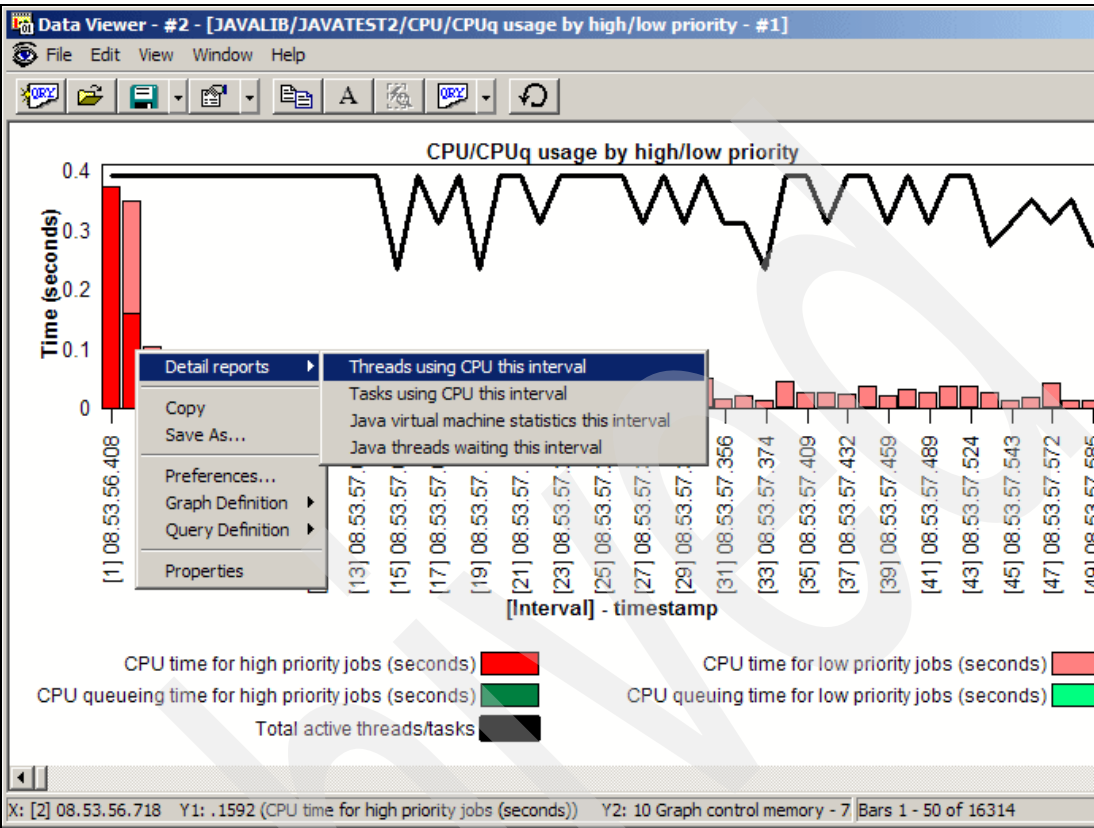


Figure 6-10 Threads using CPU this interval

Figure 6-11 shows that thread 14 clearly is using the majority of the CPU. Notice that we caught the thread in RUN status for this interval.

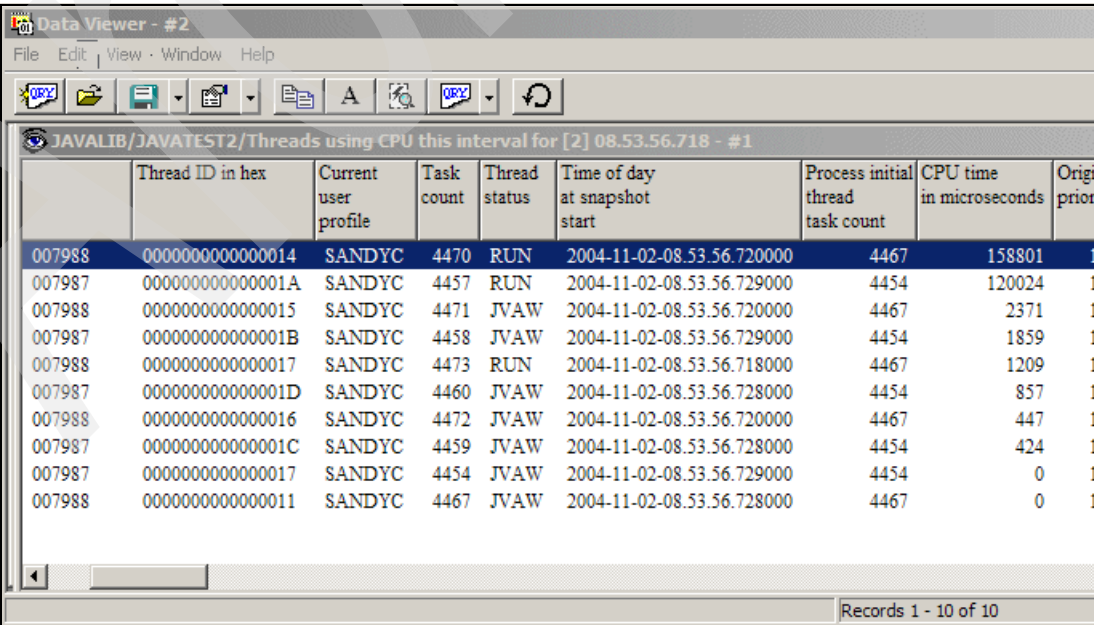


Figure 6-11 Threads using CPU this interval

Right-click on the thread that is using the most CPU in this interval and select **Properties** for further information, as shown in Figure 6-12.

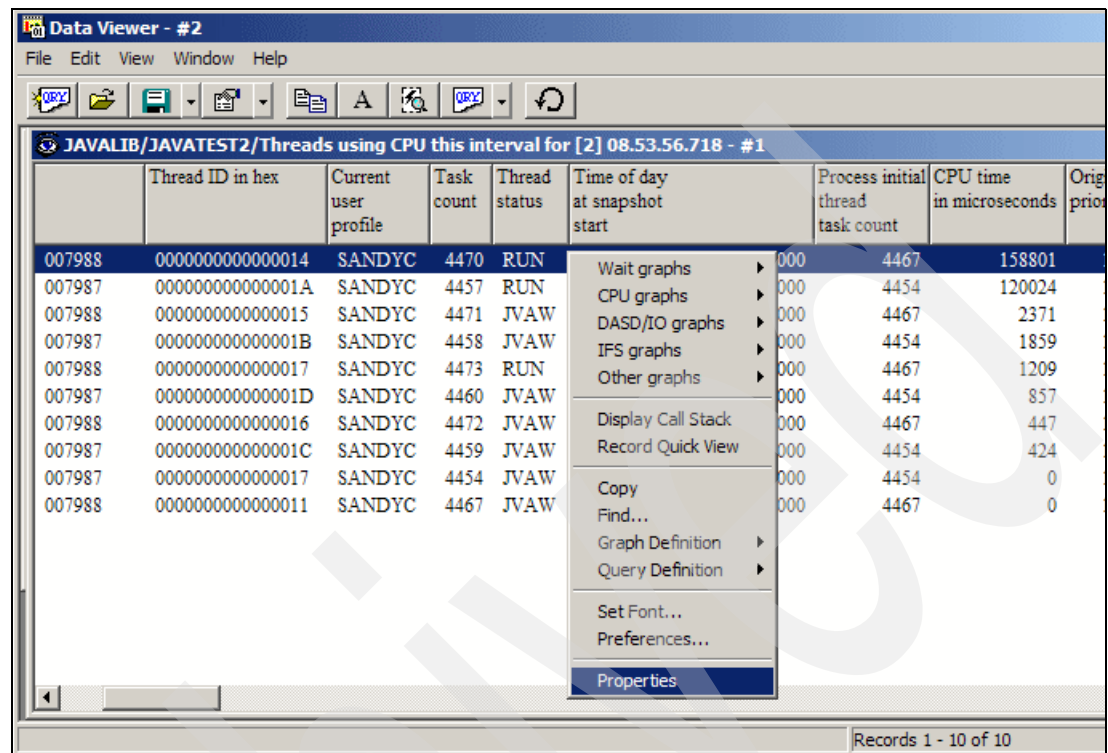


Figure 6-12 Properties for thread using most CPU this interval

We are most interested in the Call stack information shown in Figure 6-13. This is where we find out what the thread is actually doing when it is using CPU. In this example, we know that JitInterpretMethod is not the problem. This is normal Java interpretation and not the application code. The JIT is interpreting the method.

The next method we see is idocitor/testcode/lug2004/JobWatcherTest.run()V. This is a package name for an actual method and could be the source of our problem.

Keep in mind that this is one snapshot of the thousands you collected. Do not jump to conclusions based on this alone. However, if you repeat this process of looking at different intervals, a pattern may be determined.

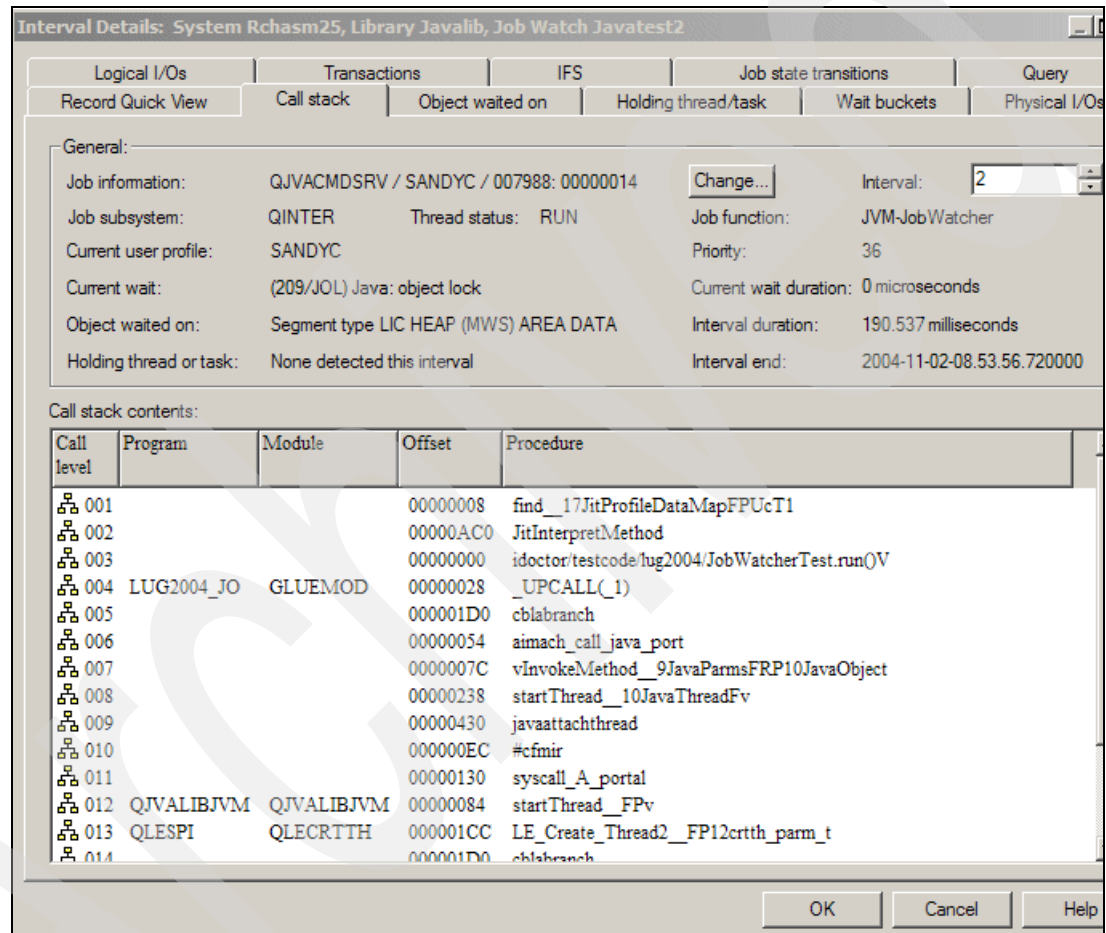


Figure 6-13 Interval Details: Call stack information

An alternative to checking individual threads one interval at a time is to use the Job Watcher RTVSTKDTA command, which we have also used in the following chapters:

- ▶ Chapter 4, “Analysis example: traditional batch ILE RPG application” on page 117
- ▶ Chapter 5, “SQL, call stack, and journal analysis examples” on page 125

RTVSTKDTA command

Use this Retrieve Call Stack Data command to export the call stack information from an internally stored format to the database file called QAIDRJWSTK. The exported data residing in QAIDJWSTK records is ready for user-written SQL queries, which we use in this chapter, as shown in Figure 6-14. See “RTVSTKDTA command” on page 250 for a description of the command and its parameters.

Retrieve Call Stack Info (RTVSTKDTA)		
Type choices, press Enter.		
Watch member name	javatest2	Name
Watch library name	javalib	Name
Module and VRM	SW53	SW52, PA53, JW53, etc.
Interval or QREC (or *ALL) . .	*all	
Task Count value (or *ALL) . . .	*all	
Debug/DUMP request	0	0=No, 1=Yes
Frame outfile member name . . .	javatest2	*NONE or member name

Figure 6-14 Retrieve Call Stack Data (RTVSTKDTA) command for Java example

After this command completes, use the Job Watcher workstation GUI session and open a Query view to run SQL statements over the QAIDRJWSTK file. Based on the information we have looked at so far (Figure 6-11 on page 167), we want to extract the stack data for task count 4470. (Recall that on the graph that identified CPU time per thread, we determined that thread 14 is Task Count 4470.) As stated in earlier chapters, the task count is how individual threads are tracked in the call stack data so we need this information to put together the query shown in Figure 6-15 on page 171.

Note: We use the Job Watcher query view (full function) interface in this SQL statement example. Job Watcher has two different SQL interfaces/environments that enable you to run SQL-based queries over your existing data. They are:

- ▶ Query definition interface (for novice SQL users)
- ▶ SQL Query view (for expert SQL users)

The SQL statement shown in Figure 6-15 is fairly simple, so you could use the Query definition interface to build the statement, if you are not familiar with SQL. See “Querying Job Watcher data” on page 222 in Appendix C, “Querying and graphing tips for Job Watcher” on page 221 for a more complete description of these SQL interfaces.

Right-click in the query view and select **Execute** or press F4 to run the query.

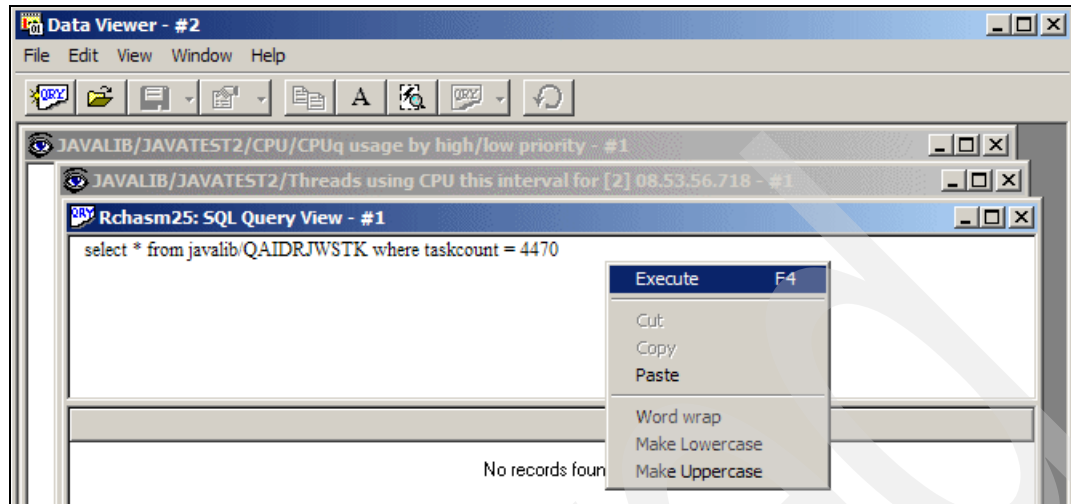


Figure 6-15 Query to extract call stack data from QAIDRJWSTK file

This query produces the information shown in Figure 6-16. Scroll through each interval of the stack data for this particular thread. We already knew that this thread is the one using the majority of the CPU; now we have a good idea of exactly what this thread was working on during this collection.

Rchasm25: SQL Query View - #1									
select * from javalib/QAIDRJWSTK where taskcount = 4470									
Interval number	Task count	Call Sta Frame Num...	Number stack frames	Stack colle reason	Program	Module	Offset	Procedure Type	Procedure Name
2	4470	2	17	S			0000>	LIC	JitInterpretMethod
2	4470	3	17	S			0000>	JAVA	idocitor/testcode/lug2004/JobWatcherTest.run()V
2	4470	4	17	S	LUG2004_JO	GLUEMOD	0000>	JAVA	_UPCALL(_1)
2	4470	5	17	S			0000>	LIC	cblabbranch
2	4470	6	17	S			0000>	LIC	aimach_call_java_port
2	4470	7	17	S			0000>	LIC	vInvokeMethod_9JavaParmsFRP10JavaObject
2	4470	8	17	S			0000>	LIC	startThread_10JavaThreadFv
2	4470	9	17	S			0000>	LIC	javaattachthread
2	4470	10	17	S			0000>	LIC	#cfmir
2	4470	11	17	S			0000>	LIC	syscall_A_portal
2	4470	12	17	S	QJVALIBJVM	QJVALIBJVM	0000>	ILE	startThread_FPv
2	4470	13	17	S	QLESPI	QLECRTH	0000>	ILE	LE_Create_Thread2_FP12crth_parm_t
2	4470	14	17	S			0000>	LIC	cblabbranch
2	4470	15	17	S			0000>	LIC	aimach_upcall_portal
2	4470	16	17	S			0000>	LIC	#aicpp
2	4470	17	17	S			0000>	LIC	pmInitiateThreadUnderTarget_Fv
3	4470	1	17	S			0000>	LIC	checkjavacastbla
3	4470	2	17	S			0000>	LIC	JitInterpretMethod
3	4470	3	17	S			0000>	JAVA	idocitor/testcode/lug2004/JobWatcherTest.run()V
3	4470	4	17	S	LUG2004_JO	GLUEMOD	0000>	JAVA	_UPCALL(_1)
3	4470	5	17	S			0000>	LIC	cblabbranch

Figure 6-16 Query results from RTVSTKDTA

At this point you should be able to put together a very good picture of the type of workload that is driving the CPU usage. If you know the application methods and stack flow that are involved and have some application knowledge, you can usually piece together the workload

or problem that is behind the high demand in CPU. If not, then you may have to go to the application developer. At least we now know what method to start with from this example and our previous call stack example: `idoctor/testcode/lug2004/JobWatcherTest.run()`V.

6.3 Heap growth

Excessive heap growth is a common reason for a Java application performance problem. Heap growth may be caused by your application having a memory leak or of the garbage collection not doing its job. Memory leaks in the Java language are a significant contributor to garbage collection bottlenecks. If you do not manage garbage collection, it can have a significant negative impact on application performance, especially when running on symmetric multiprocessing (SMP) server machines. The i5/OS Java Virtual Machine (JVM) uses concurrent (asynchronous) garbage collection. This type of garbage collection results in shorter pause times and enables application threads to continue processing requests during the garbage collection cycle.

Some symptoms of heap growth include:

- ▶ Poorly performing application that may have a leak
- ▶ Heavy page faulting in the storage pool where your WebSphere application runs (Java does not tolerate heavy paging)

You can use several tools to evaluate a possible heap growth problem:

- ▶ The Dump Java Virtual Machine (DMPJVM) command

Important: The DMPJVM command requires the job to be held in order to collect heap object table statistics. This command does not work on some excessively large WebSphere heaps. iDoctor for iSeries Heap Analyzer runs in the background and will run on enormous heaps without any affect on the JVM operation while it is running.

- ▶ Start Service Tools (SST)
- ▶ Work with System Status (WRKSYSSTS) to investigate faulting and the wait-to-ineligible transitions.
- ▶ iDoctor for iSeries Heap Analyzer

6.3.1 Heap Analyzer

Job Watcher is very effective at investigating the JVM operations at a thread level; however, to investigate the garbage collector and heap growth, Heap Analyzer is a more effective tool. The Heap Analyzer component of the iDoctor for iSeries tools is used to perform Java application heap analysis and object creation profiling (size and identification) over time.

For our example, we assume that your Java application, running under WebSphere in this case, is experiencing heap growth. We need the same information whether that growth extends over days or hours. The only thing that changes is the timing of the collection.

Heap Analyzer includes a call stack for every snapshot when run in the profile mode so the created objects can be mapped to the application functions.

As with Job Watcher, you access the Heap Analyzer from the iDoctor Components window shown in Figure 6-17. Click **Launch**.

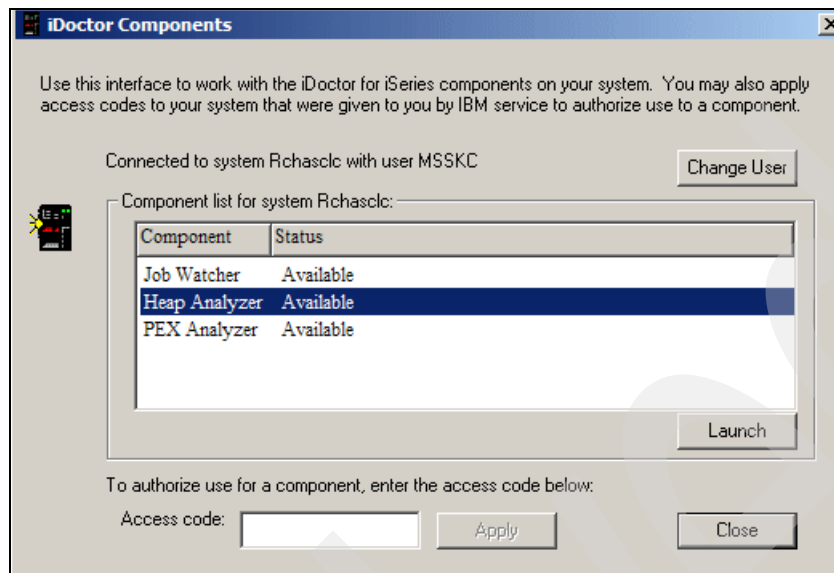


Figure 6-17 iDoctor Components: Heap Analyzer

Right-click **Heap Analyzer** → **Start Heap Watch** (Figure 6-18).

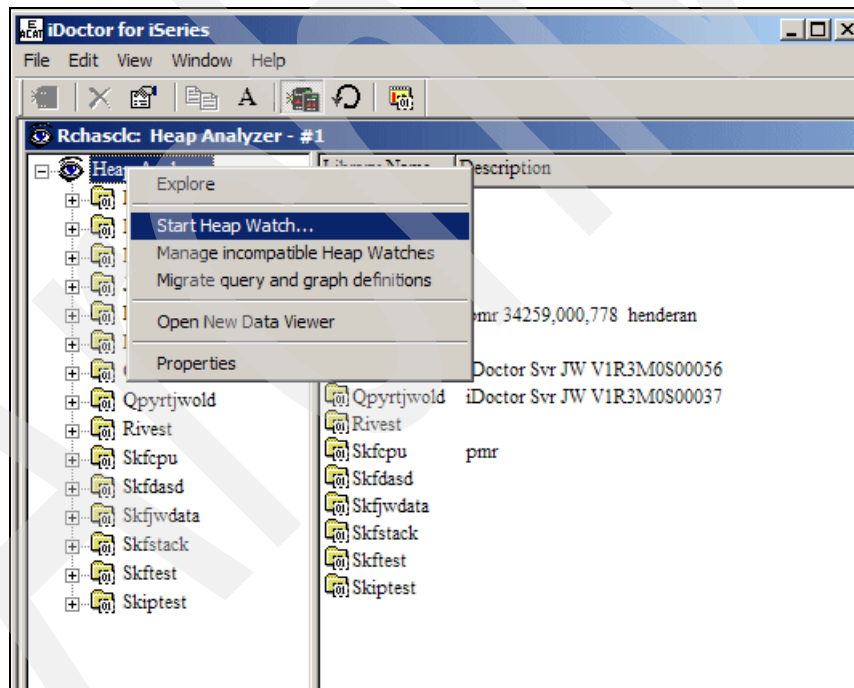


Figure 6-18 Start Heap Watch

There are three phases to the Heap Watch, as shown in Figure 6-19 on page 174:

1. Object table dump: produces a dump of the live object table
2. Object create profile: profiles the object creates per thread and per object size
3. Object root finder: does detailed analysis on a specific object to identify where it is rooted in the Garbage Collection (GC) heap.

Object table dump

It is very important to collect at least one Object table dump in a Java environment—before a problem actually surfaces—with the Java Virtual Machine (JVM) active. This helps to establish a baseline to compare any future dumps. After you see that the heap is showing growth, you want several object table dumps to use for establishing a pattern of growth and identifying the object class type and the size of the object you feel may be causing the growth.

The frequency of the dumps depends on how fast the leak is. For example, if you have heap growth at 200 megabytes an hour, then a dump every two hours might be the right choice. If your growth is more like 1 GB an hour, then it would be better to get a dump at least every 45 minutes. These dumps are relatively small in size so space should not be an issue. The main concern is that we want the data before any heavy storage management paging starts.

1. The first Heap Watch Wizard window (Welcome) presents three choices for starting Heap Watch, as shown in Figure 6-19. Select **Object table dump**. Click **Next**.

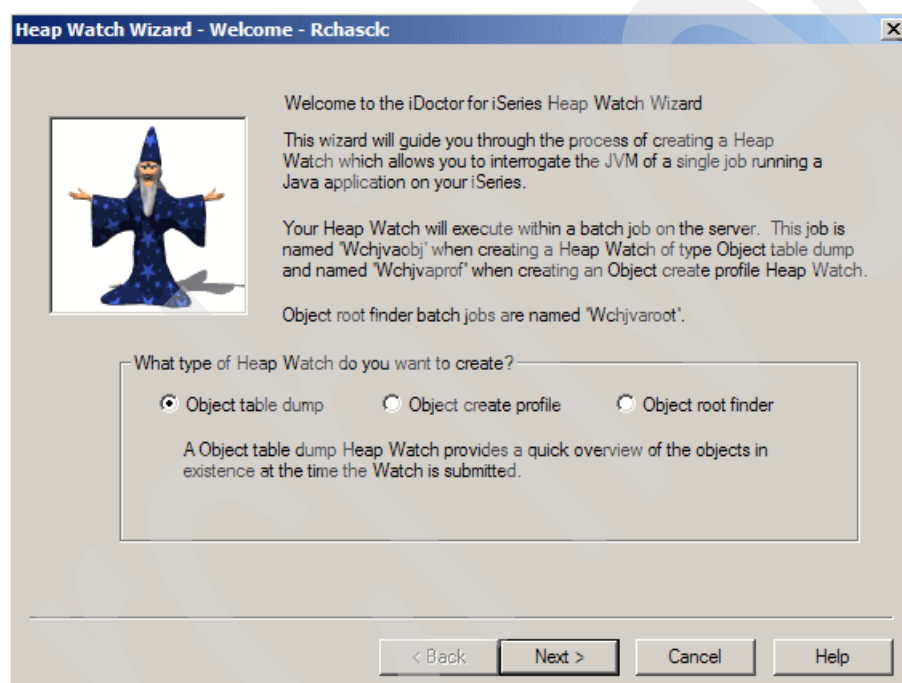
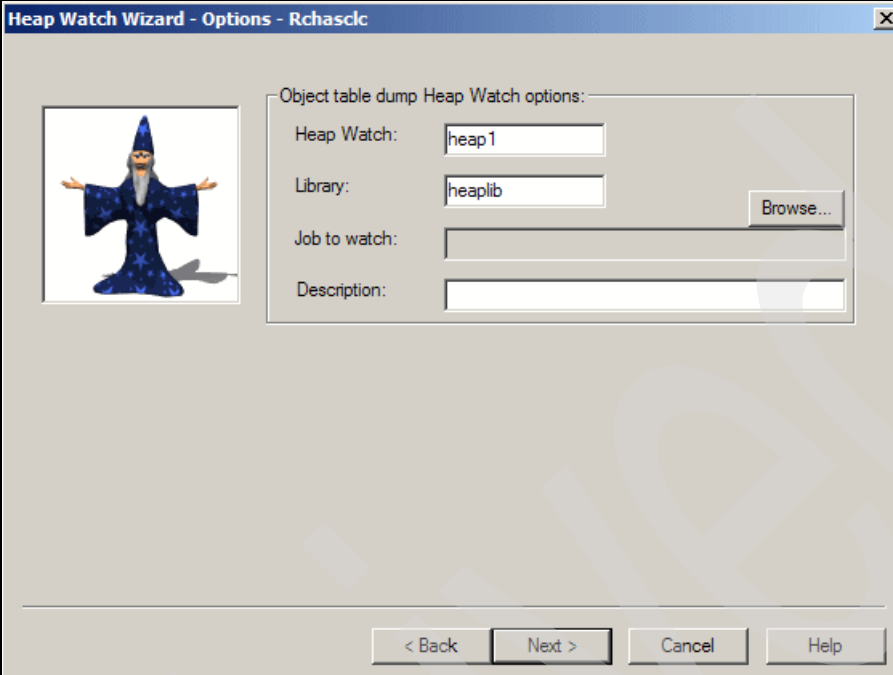


Figure 6-19 Heap Watch Wizard: Welcome

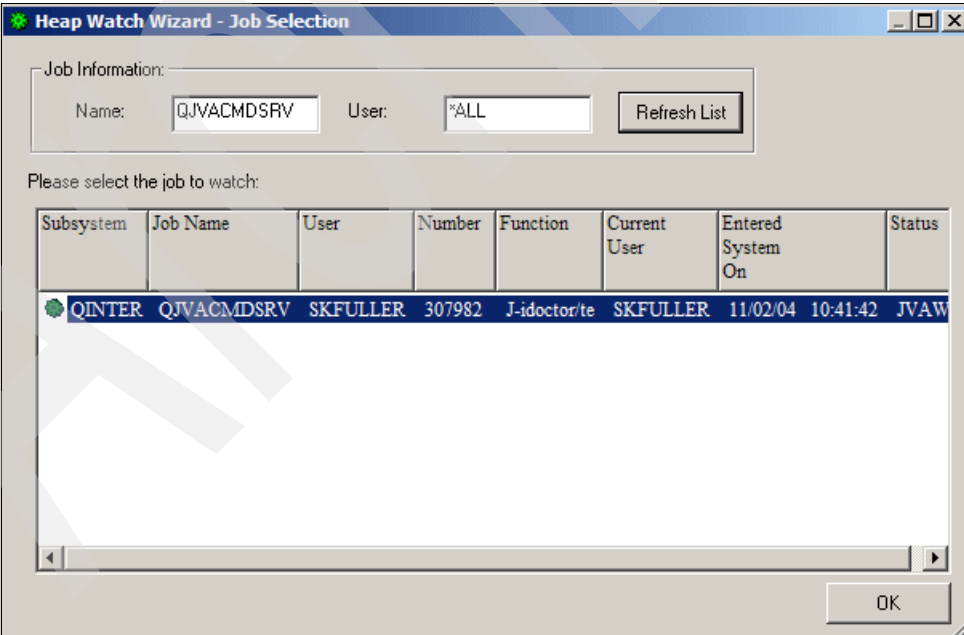
2. Fill in the Heap Watch options with a name and library to identify the watch being started, as shown in Figure 6-20. Click **Browse** to find an existing job to be watched.



The dialog box is titled "Heap Watch Wizard - Options - Rchasc.c". It features a wizard icon of a wizard on the left. The main area is titled "Object table dump Heap Watch options:" and contains four input fields: "Heap Watch:" with the value "heap1", "Library:" with the value "heaplib", "Job to watch:" (empty), and "Description:" (empty). A "Browse..." button is located to the right of the "Library:" field. At the bottom, there are four buttons: "< Back", "Next >", "Cancel", and "Help".

Figure 6-20 Heap Watch: options

3. The default filter for the job selection screen is QJVACMDSRV, as shown in Figure 6-21. Make changes here as needed. Select the job you are interested in. Click **OK**.



The dialog box is titled "Heap Watch Wizard - Job Selection". It has a "Job Information:" section with "Name:" set to "QJVACMDSRV" and "User:" set to "ALL". A "Refresh List" button is next to the "User:" field. Below this, it says "Please select the job to watch:" followed by a table of jobs. The table has columns: Subsystem, Job Name, User, Number, Function, Current User, Entered System On, and Status. One row is highlighted with a green background.

Subsystem	Job Name	User	Number	Function	Current User	Entered System On	Status
QINTER	QJVACMDSRV	SKFULLER	307982	J-idocotr/te	SKFULLER	11/02/04 10:41:42	JVAW

An "OK" button is located at the bottom right of the dialog box.

Figure 6-21 Heap Watch Wizard: Job Selection

4. Optionally, you can set up a time limit or other options for ending the table dump watch as shown in Figure 6-22. Typically, we do not recommend setting a time limit because you may get a partial set of data, so **None** is selected. Click **Next**.

This command has very little overhead and does not affect the JVM. Let it run as long as it needs to complete. Even if it takes several minutes to run, we would rather have a complete set of data than partial data.

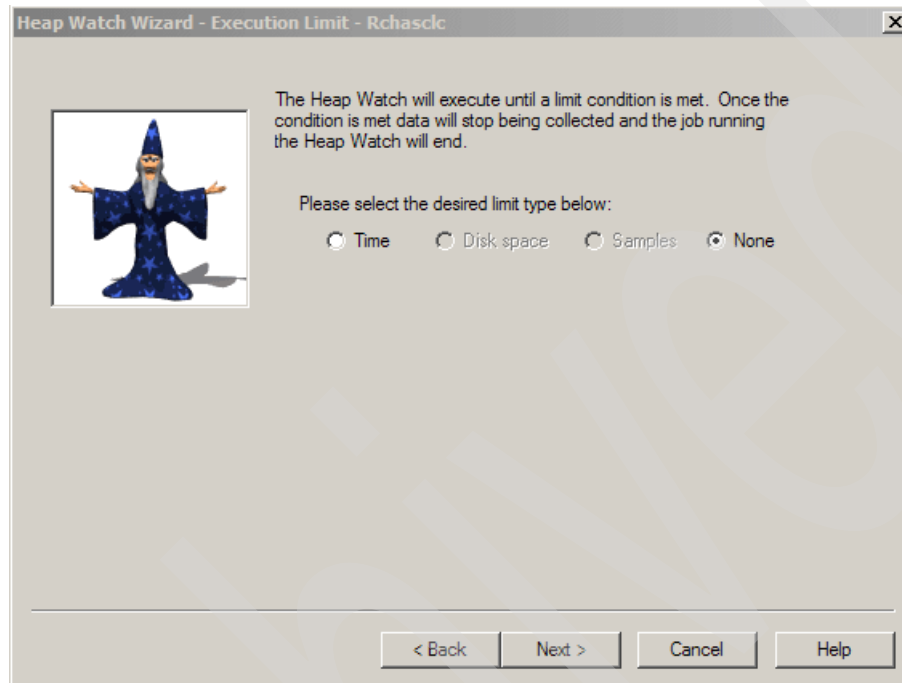


Figure 6-22 Heap Watch Wizard: Execution Limit

- Review the set-up options as shown in Figure 6-23, and click **Finish** to start the Heap Watch for the Object Table Dump.

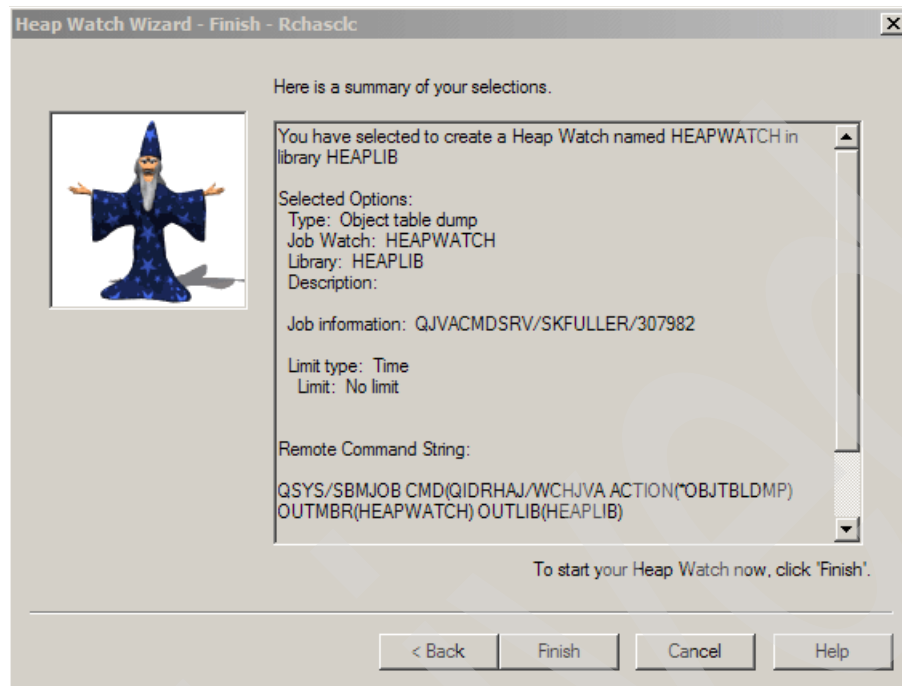


Figure 6-23 Heap Watch Wizard: Finish

- Return to the main iDoctor panel. Right-click the collection library and select **Explore** to see the watch status. When the watch has completed, select **Explore** to investigate the data. See Figure 6-24.

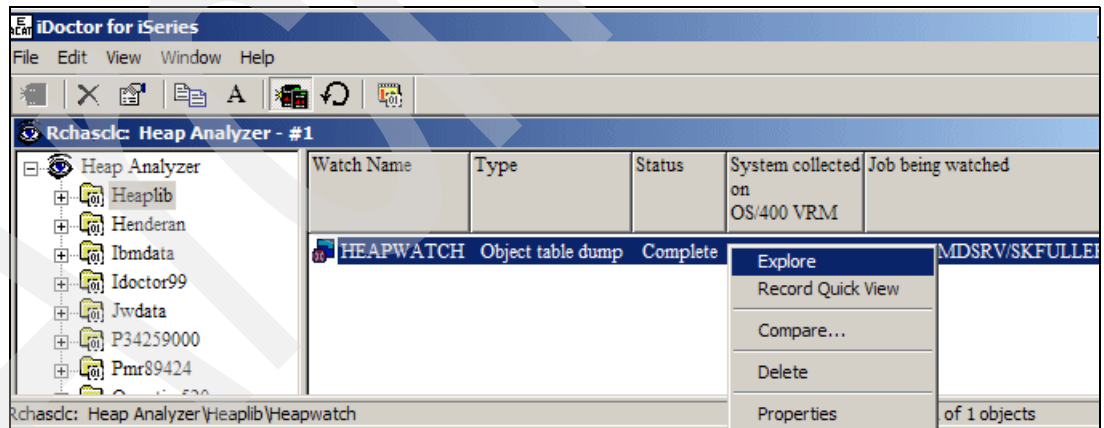


Figure 6-24 Heap Watch complete status

7. Select the iDoctor-supplied queries to begin with. Right-click and select **Explore** as shown in Figure 6-25.

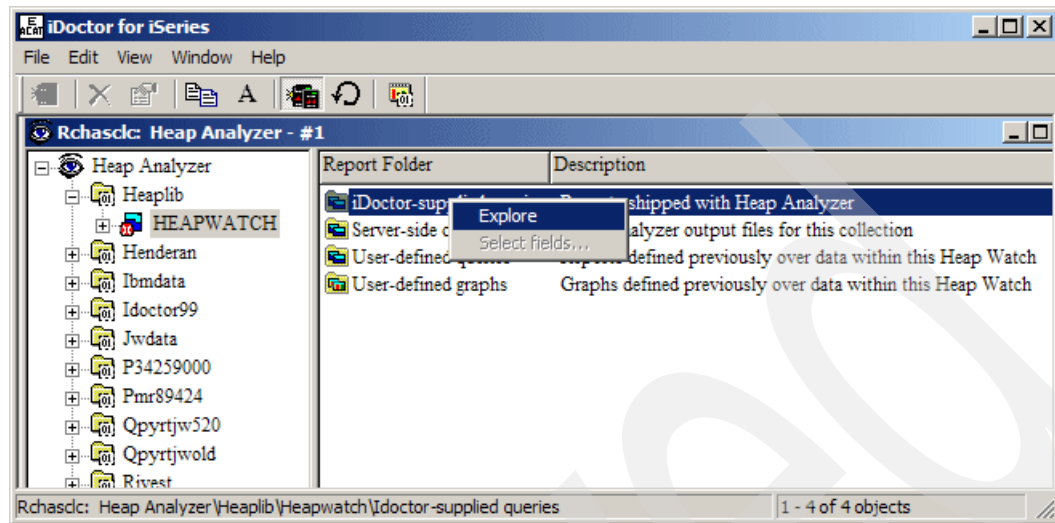


Figure 6-25 Explore iDoctor-supplied queries

8. There is one view available, as shown in Figure 6-26. It is basically a full report of all live objects in the heap. Right-click the description and select **Open Table(s)** to see the report.

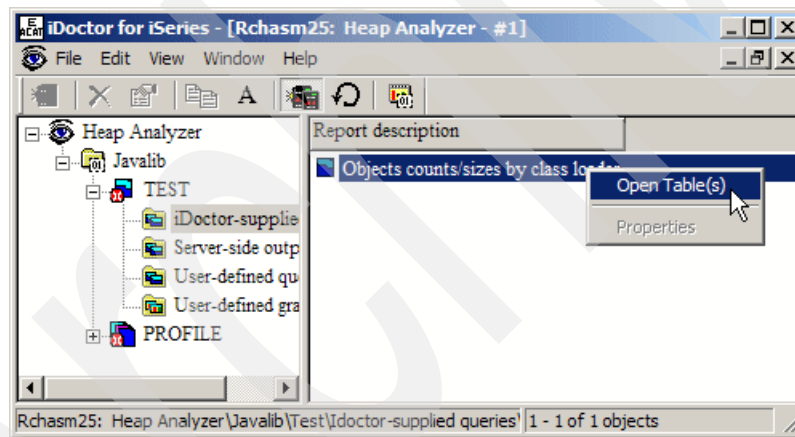


Figure 6-26 Open table of Objects counts/sizes by class loader report

- Click twice on the column heading **Total Objs Heap Size** to order the information according to the amount of space used. This gives you a quick look at what live objects are contained in the heap. Often something interesting will surface from this if you are having heap growth problems.

In the example shown in Figure 6-27, you would certainly pick the java/lang/Error object class objects to start with even though this entry is not on the top of the list.

Class Loader Name	Object Class Name	Object Count	Total Objects Size (bytes)	Total Objs Heap Size (bytes)
default	[J	2741217	219433824	219530032
default	[Ljava/lang/Object;	59	201953392	202257600
default	java/lang/Error	2741231	153508936	175438784
sun/misc/Launcher\$AppClassLoader	idoctor/testcode/lug2004/Jobdata	24278	582672	776896
default	java/lang/String	3089	135916	437504
default	[B	73	311766	334240
default	[Ljava/lang/String;	1584	92480	97072
default	[C	511	75068	90656
default	java/lang/Class	662	68848	84736
default	java/util/Hashtable\$Entry	605	31460	38720
default	[[Ljava/lang/String;	6	11208	16448
default	java/util/HashMap\$Entry	247	12844	15808
default	[Ljava/util/Hashtable\$Entry;	21	10544	14816
default	java/util/TreeMap\$Entry	136	9248	10880
default	[Ljava/util/HashMap\$Entry;	25	6152	10160
default	java/net/URL	73	8176	9344
default	java/lang/ref/Finalizer	66	5808	6336

Figure 6-27 Objects counts/sizes by class loader report

Note: In a Heap Watch Object Class Name column, as shown in Figure 6-27, you see several entries where the first character is a left bracket symbol: [. The left bracket is used to indicate an array object, for example, [J.

Note that you may need to investigate whether growth in one object type might be a side effect of growth in another object type.

Sometimes, just identifying the object that is growing or leaking is enough to suggest corrective action. However, it is often necessary to go to the next level of analysis: to identify where the object creates are coming from. This is particularly true in the case where the object identified is a generic Java object such as String. In our example, we want to know where the java/lang/Error objects are being created.

This leads us to using the Heap Watch *Object create profile* function you saw as an option in the earlier Start Heap Watch wizard window.

Object create profile

Another good way to understand heap issues is to turn on the Object create profile during a period where you are seeing the heap growth. We have identified an object class we are interested in from the Object table dump described in “Object table dump” on page 174, so

we normally use the profiler output to learn where the object is being created in the application.

1. To investigate our example further using the Object create profile, we need to start another Heap Watch. (If necessary, refer to Figure 6-18 on page 173.) This time we select **Object create profile** as shown in Figure 6-28. Click **Next**.

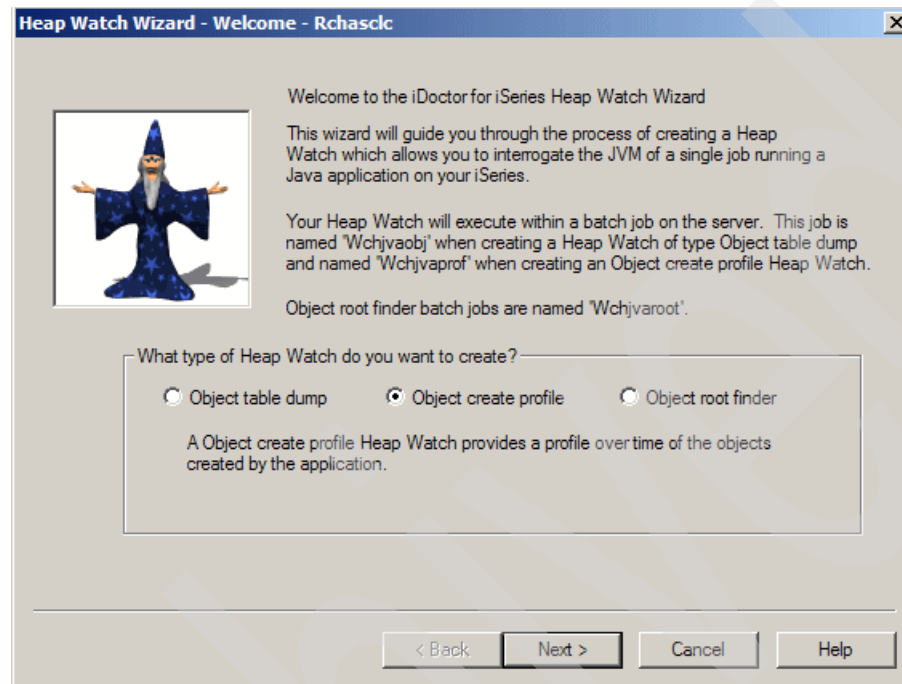


Figure 6-28 Heap Watch Wizard: Welcome for Object create profile

2. Figure 6-29 shows the profiler options. Fill in the Heap Watch options with a name and library to identify the watch. Click **Browse** to identify the job to watch. Select the same job as before.

Heap Watch Wizard - Options - Rchasm25

Object create profile Heap Watch options:

Heap Watch: PROFILE

Library: JAVALIB Browse...

Job to watch: QJVACMSRV/SANDYC/008110

Description:

Object create profile advanced options:

Initiate PEX collection for Java(*SERVICE): ☒ Yes ☐ No

Invocation stack format: Single column

Additional stack programs to skip: 5 1 - 50, *NONE

Pre-allocate output member records: *NONE 1 - 9999999, *NONE, *SAMPLES

< Back Next > Cancel Help

Figure 6-29 Create object profile options

Consider the Object create profile advanced options:

- ▶ **Initiate PEX collection** almost always should be **Yes**. The only exception is if you already had a PEX collection running on your iSeries system. This is just a switch that enables the collection of the object creates.
- ▶ **Invocation stack format** should always be **Single column**.
- ▶ The **Additional stack programs to skip** parameter enables you to influence what part of the stack you collect. For performance reasons we only collect five frames. If you do not skip any, then the last five frames on the stack are almost always IBM code involved with the tracing. So, typically you skip at least four frames, which means set this value to 5.
- ▶ **Pre-allocate output member records** is a performance parameter for the collection. If you pre-allocate, you can avoid some long times between intervals.

Click **Next**.

3. As before, there are three ways to limit the collection:

- Time (in seconds)
- Disk Space (in megabytes)
- Samples (by count)

For our example, we choose a **Time** limit for the collection (Figure 6-30). Click **Next** to continue.

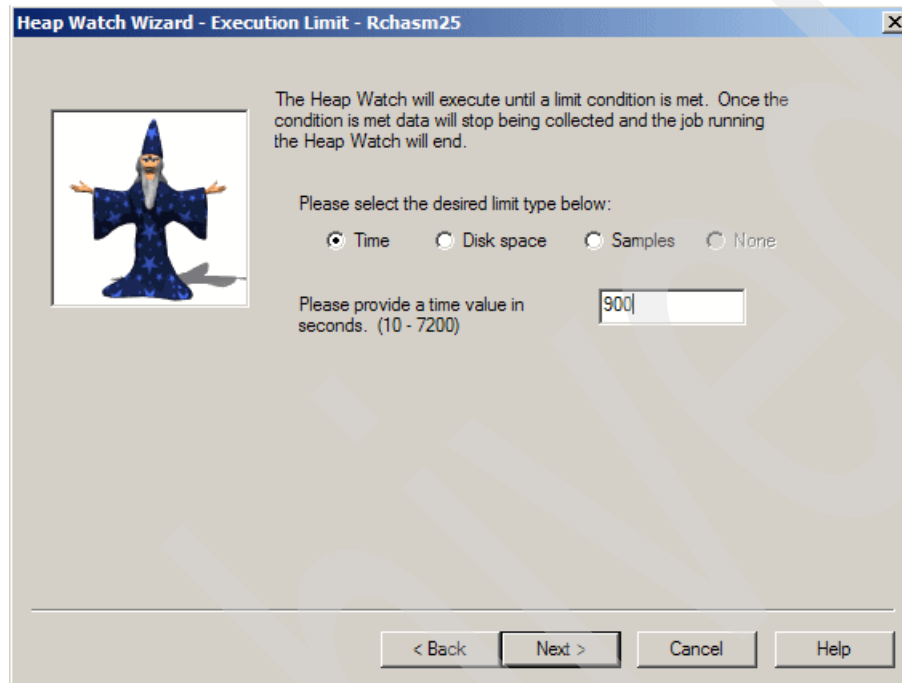


Figure 6-30 Object create profile execution limit

Note: Specifying a long collection time may cause a noticeable increase in disk space utilization on your iSeries system.

4. Review the summary of your selections as shown in Figure 6-31. Click **Finish** to submit.

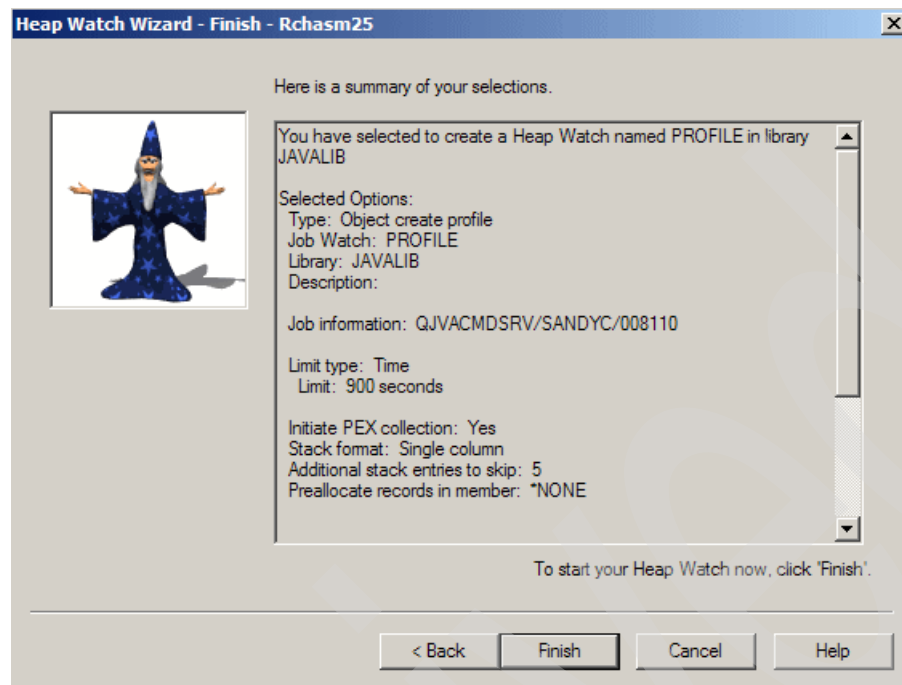


Figure 6-31 Create profile object summary

5. After the Object create profile has completed, right-click it and select **Explore** to investigate the data.

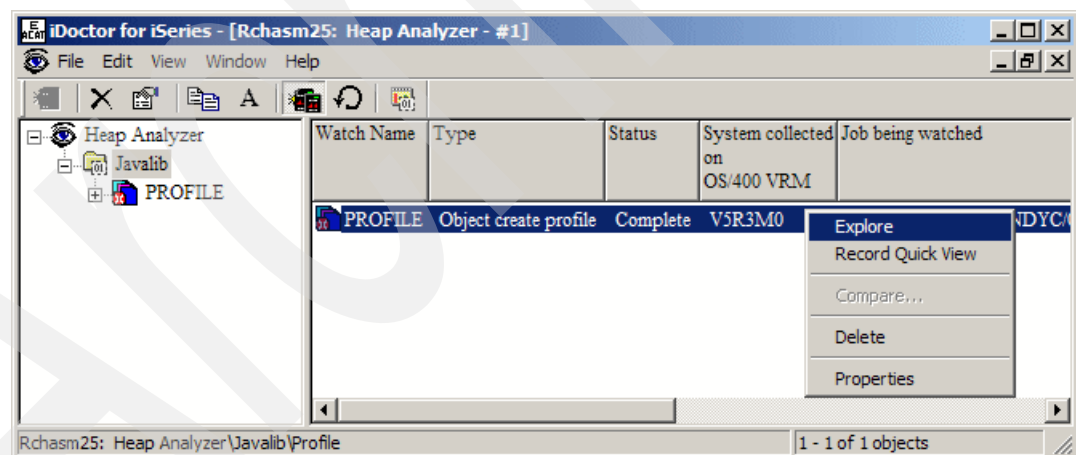


Figure 6-32 Explore Object create profile

6. Select the iDoctor-supplied queries to begin with. Right-click and select **Explore** as shown in Figure 6-33.

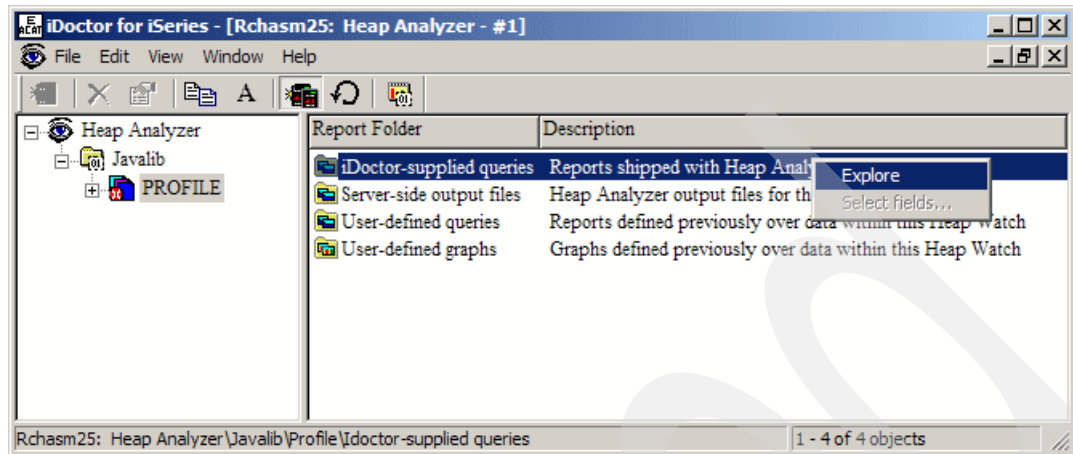


Figure 6-33 Explore iDoctor supplied queries

7. The Objects created per interval profile is a good report to start with because, in this example, we already know the object Class Name we are looking for. Select **Open Table(s)** as shown in Figure 6-34.

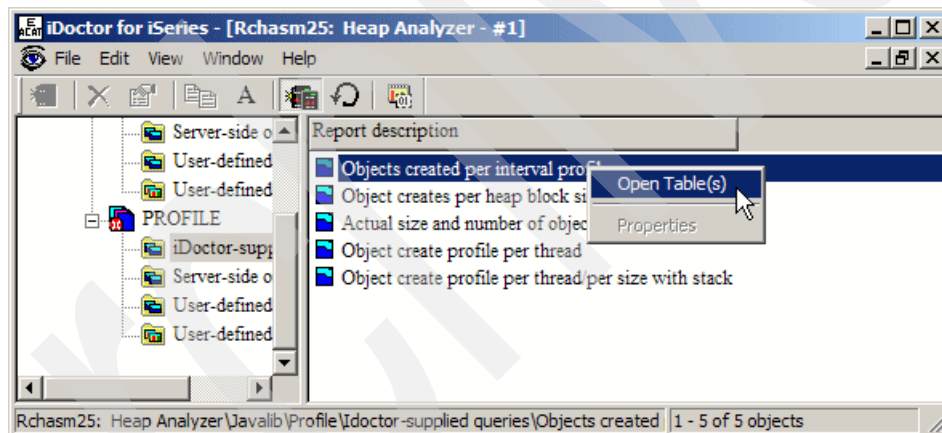


Figure 6-34 Open Objects created per interval profile report

- Figure 6-35 shows the list of objects created in our profile. Select one of the Object instances, and all the information is grouped by interval. The interesting thing here is the call stack data, which shows where the object create came from within the application.

Interval Number	Objects Created This Size	Object Size	OBJNAME	Call Stack
1645251	44	56	java/lang/Error	^~NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2
1645251	44	144	[J	^~InterpretJavaMethod_15JavaInterpreterFP11method_infoP25JavaInterpreterStackFrame
69001	42	24	idoctor/testcode/lug2004/Jobdata	^~GLUEMOD/*PROC/TESTCODE_LUG2004_JOBDATA*_INTERPRET_SL8(LL)V
69001	42	56	java/lang/Error	^~InterpretJavaMethod_15JavaInterpreterFP11method_infoP25JavaInterpreterStackFrame
69001	42	144	[J	^~GLUEMOD/*PROC/LUG2004_JOBWATCHERTEST*_INTERPRET_SL16(L)V
2010001	39	24	idoctor/testcode/lug2004/Jobdata	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
2010001	39	56	java/lang/Error	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
2010001	39	144	[J	NewArray_13JavaNewObjectF8JavaTypelQ2_13JavaNewObject11Callers.Mode buildSta
5474751	39	24	idoctor/testcode/lug2004/Jobdata	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
5474751	39	56	java/lang/Error	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
5474751	39	144	[J	NewArray_13JavaNewObjectF8JavaTypelQ2_13JavaNewObject11Callers.Mode buildSta
6325751	36	24	idoctor/testcode/lug2004/Jobdata	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
6325751	36	56	java/lang/Error	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
6325751	36	144	[J	NewArray_13JavaNewObjectF8JavaTypelQ2_13JavaNewObject11Callers.Mode buildSta
4411001	34	24	idoctor/testcode/lug2004/Jobdata	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
4411001	34	56	java/lang/Error	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
4411001	34	144	[J	NewArray_13JavaNewObjectF8JavaTypelQ2_13JavaNewObject11Callers.Mode buildSta
3377501	33	24	idoctor/testcode/lug2004/Jobdata	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn
3377501	33	56	java/lang/Error	NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2 InterpretJavaMethod_15JavaIn

Figure 6-35 Objects created per interval profile

- Go back to the Report descriptions shown in Figure 6-34 on page 184 and select the **Object create profile per thread/per size with stack** view to see the same type of information rolled up at the thread level for particular object sizes (Figure 6-36).

Thread's Task Count	Heap Block Size (bytes)	Object Size	OBJTOTAL	Call Stack
13553	192	144	91818	NewArray_13JavaNewObjectF8JavaTypelQ2_13JavaNewObject11Callers.Mode buildStack
13553	64	56	91818	^~NewObjectS_Pdc_13JavaNewObjectFP9JavaClassUIT2
13553	32	24	91817	^~InterpretJavaMethod_15JavaInterpreterFP11method_infoP25JavaInterpreterStackFrame
13555	192	144	53144	^~GLUEMOD/*PROC/TESTCODE_LUG2004_JOBDATA*_INTERPRET_SL8(LL)V
13555	32	24	53144	^~InterpretJavaMethod_15JavaInterpreterFP11method_infoP25JavaInterpreterStackFrame
13555	64	56	53144	^~GLUEMOD/*PROC/LUG2004_JOBWATCHERTEST*_INTERPRET_SL16(L)V

Figure 6-36 Object create profile per thread/per size with stack

The key thing to remember here is that this is a profiler. As with any profiling data, the longer the collection runs, the more accurate your data will be. Look for things that represent patterns and enable you to draw conclusions from the data. In Figure 6-36, we find that the objects come from the method `JOBDATA` within the application.

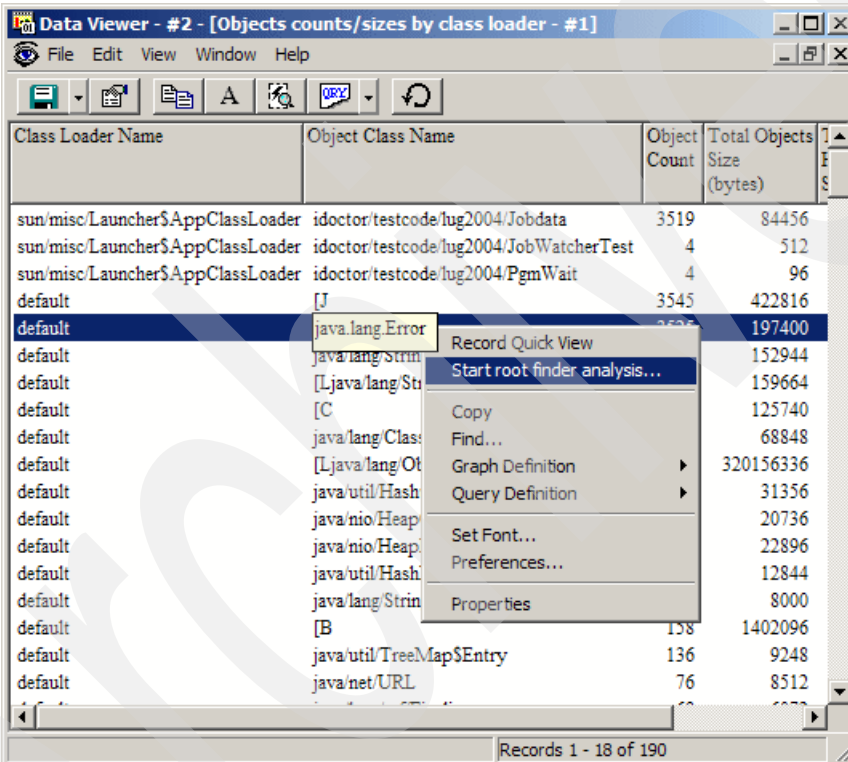
So far we know what objects are accounting for the heap growth. We also know where (method JOBDATA) in the application the object creates are coming from. In many cases this is sufficient information to have a discussion with the application developers and begin to investigate potential code issues.

However, in some very complex scenarios it can still leave a question about why these objects are not being collected (cleaned up and removed from the heap by garbage collection). In this situation, the next step is to use the Object root finder.

Object root finder

To use the Object root finder, go back to your Object table dump access and select **Object counts/sizes by class loader** as shown in Figure 6-26 on page 178. Select **Open Table(s)**.

1. Using Figure 6-37 as an example, right-click on any object of interest and select **Start Root Finder analysis**.



The screenshot shows a window titled "Data Viewer - #2 - [Objects counts/sizes by class loader - #1]". It contains a table with the following columns: "Class Loader Name", "Object Class Name", "Object Count", and "Total Objects Size (bytes)". The table lists various objects and their counts and sizes. A right-click context menu is open over the row for "java.lang.Error", with the option "Start root finder analysis..." highlighted.

Class Loader Name	Object Class Name	Object Count	Total Objects Size (bytes)
sun/misc/Launcher\$AppClassLoader	idocor/testcode/lug2004/Jobdata	3519	84456
sun/misc/Launcher\$AppClassLoader	idocor/testcode/lug2004/JobWatcherTest	4	512
sun/misc/Launcher\$AppClassLoader	idocor/testcode/lug2004/PgmWait	4	96
default	[J	3545	422816
default	java.lang.Error	2532	197400
default	java.lang.String		152944
default	[Ljava.lang.String		159664
default	[C		125740
default	java.lang.Class		68848
default	[Ljava.lang.Object		320156336
default	java.util.HashMap		31356
default	java.nio.HeapByteBuffer		20736
default	java.nio.HeapByteBuffer		22896
default	java.util.HashMap		12844
default	java.lang.String		8000
default	[B	158	1402096
default	java.util.TreeMap\$Entry	136	9248
default	java.net.URL	76	8512

Figure 6-37 Start root finder analysis

With this support we do some very intense processing on the heap to investigate how the objects are being used. For this operation to work, it has to be done on an active JVM. In fact, any analysis for the root finder has to be done when the JVM is active. This differs from the Object table dump and Object create profile in that with those tools we can collect data on an active JVM then perform analysis on that data after the JVM has gone away.

2. When the root finder is started by right-clicking on a class name as we have done, the Heap Watch Wizard options window opens with the correct information filled in for JVM handle and Class handle as shown in Figure 6-38.

The only time you would have an Object handle instead of a Class handle is if you choose to run the root finder against a root finder collection. For instance, if you ran root finder by selecting a class, the code randomly selects an individual object to investigate. From there you might find that object A is anchored in object B. You can then select to run the root analysis specifically on object B. In this case, your options screen will have JVM and Object handle values filled in instead of JVM and Class handles.

If you ran previous Heap Watch iterations, you have the option to remove those. Click **Next** to continue.

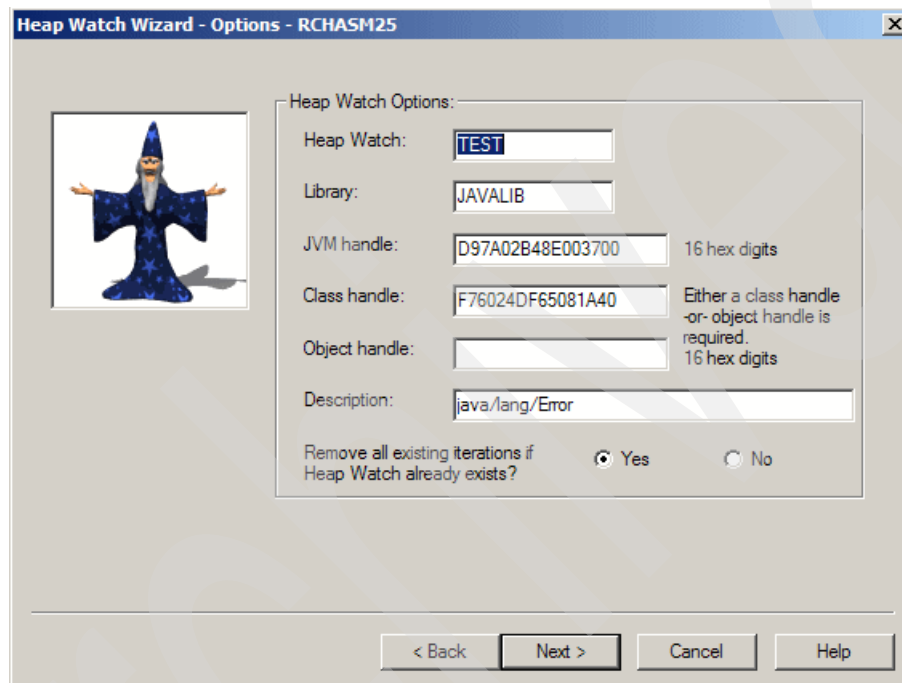


Figure 6-38 Heap Watch Wizard: Options for Root Finder Analysis

3. The root finder processing is very intense but does not involve significant use of disk space. For that reason, the only execution limit available is time (Figure 6-39). Click **Next**.

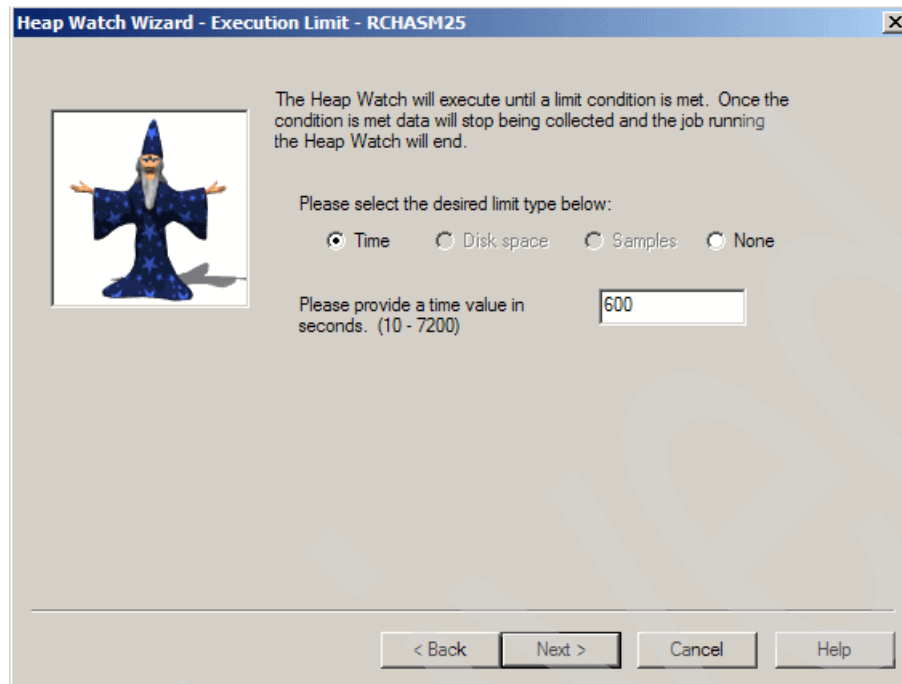


Figure 6-39 Root finder execution limit

4. Review the summary of your selections (Figure 6-40). Click **Finish** to start the Object root finder collection.

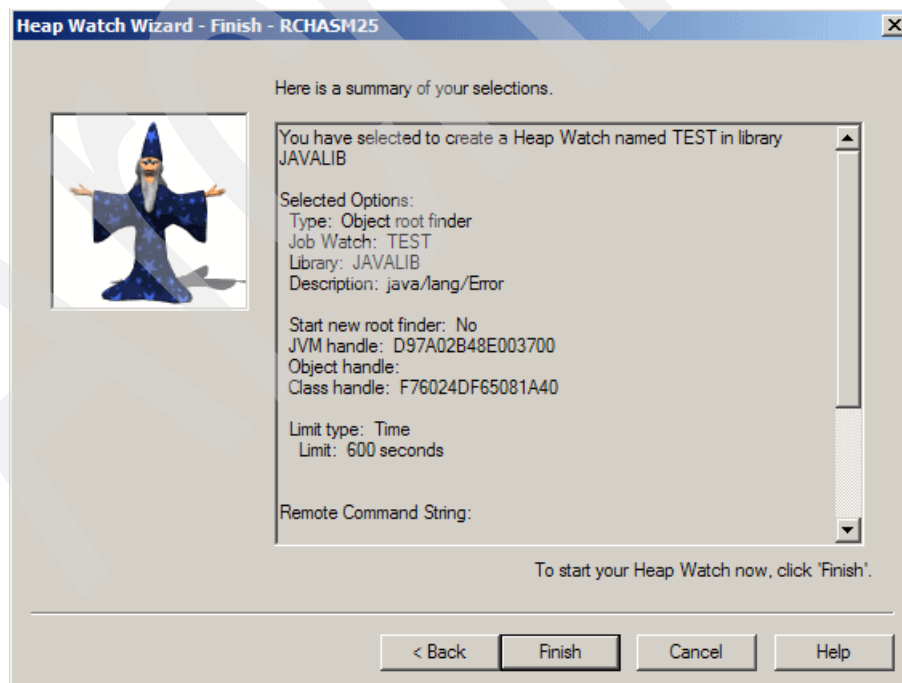


Figure 6-40 Root finder summary

- When the Object root finder collection has completed, double-click it to open the tree view of the data displayed in the right-side pane. Right-click **Object trees by iteration** and select **Explore** as shown in Figure 6-41.

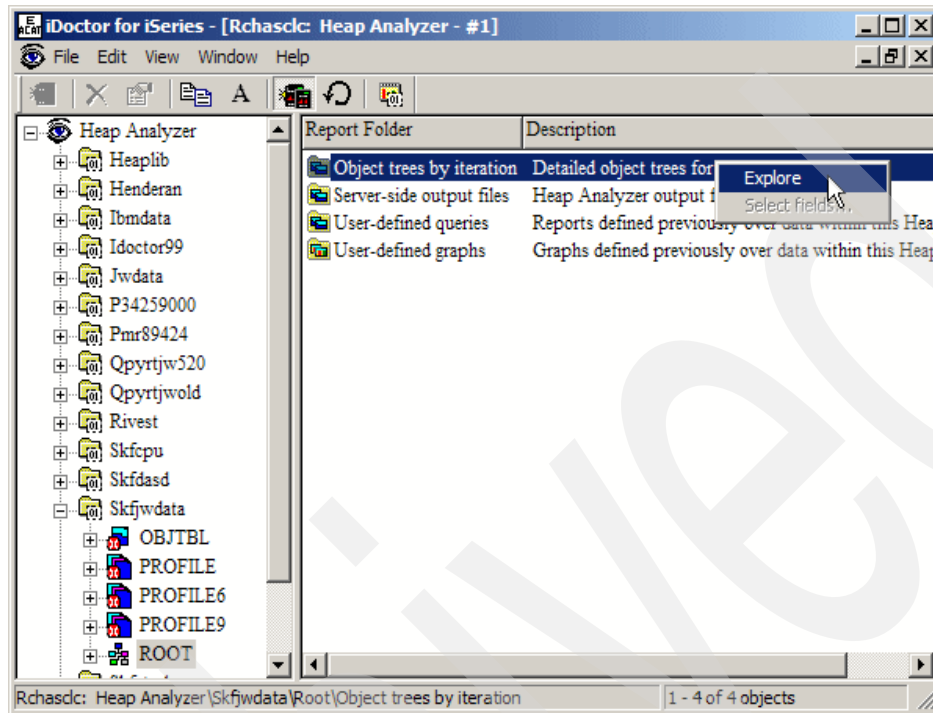


Figure 6-41 Explore Object trees by iteration

- Figure 6-42 shows the iteration. Right-click the iteration and select **Open Tree(s)**.

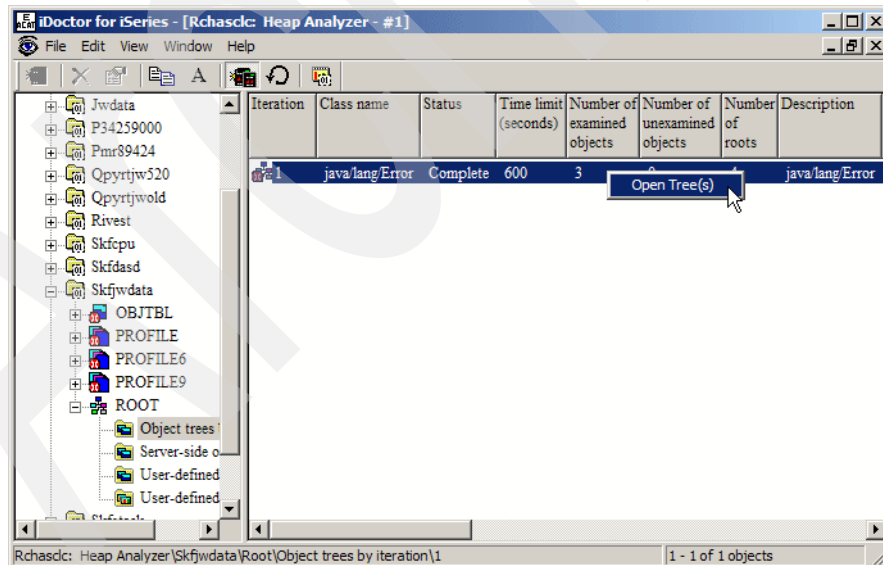


Figure 6-42 Open Object trees by iteration

The initial object tree view before expansion is shown in Figure 6-43.

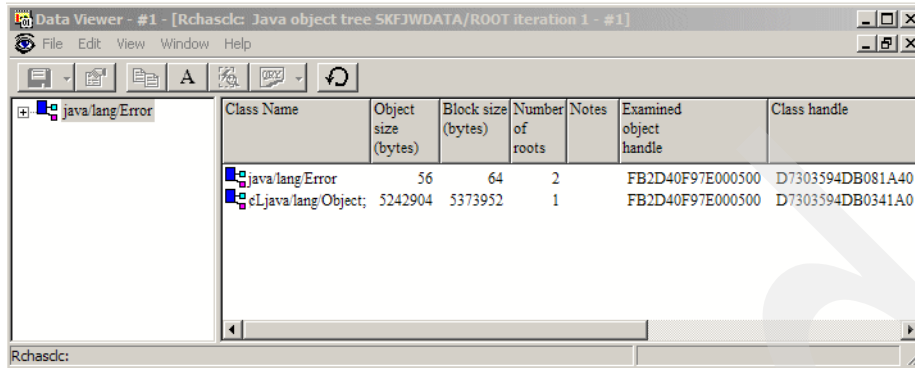


Figure 6-43 Java object tree root iteration before expansion

7. Fully expand the tree in the left pane to see the relationships and the roots (Figure 6-44). From this example, we see that the java/lang/Error object is anchored in a vector.

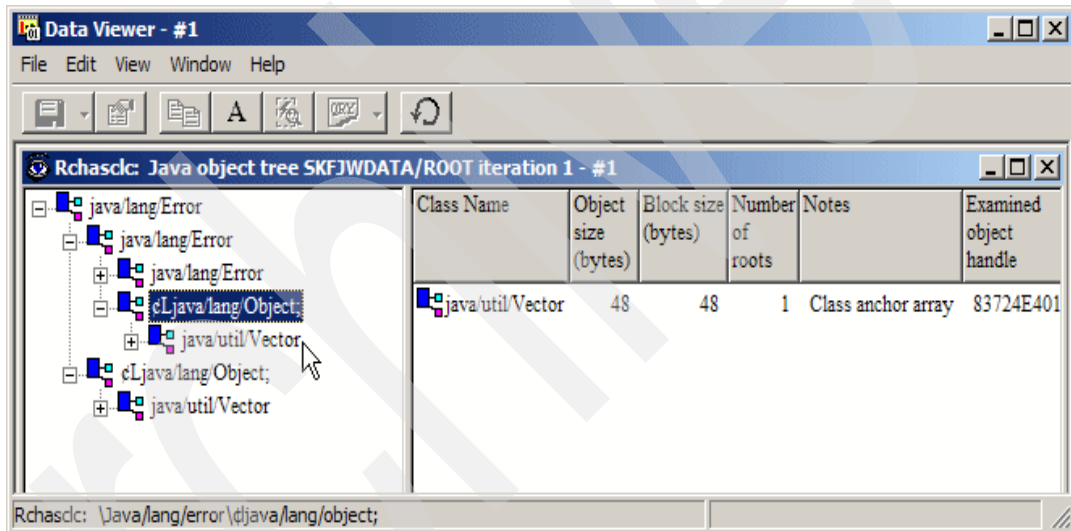


Figure 6-44 Expanded Java object tree root

Because the application is putting them into a vector, they are not getting cleaned up by the garbage collector. For any further investigation, you have to go to the application source code or the developer. We know from previous examples that the source to investigate is the JOBDATA method.



Installing and uninstalling Job Watcher details

This appendix gives you the detailed install and uninstall steps for Job Watcher.

It includes information about:

- ▶ iSeries installation requirements
- ▶ PC installation requirements
- ▶ PTF prerequisites
- ▶ Steps to install and uninstall iDoctor for iSeries

Installation requirements

The installation of Job Watcher consists of:

- ▶ Client server code (PC requirements)
 - Windows NT® 4.0, 2000, XP
 - IBM iSeries Access for Windows V5R3
- ▶ iSeries server code
 - IBM i5/OS V5R3
 - PTFs MF33560, SI15095
 - The user profile performing the installation must have *SECOFR user class and special authorities *ALLOBJ and *SECADM
 - The following host servers (identified by the SERVER parameter values on the STRHOSTSVR command) must be running on the server:
 - *DATABASE
 - *RMTCMD
 - *SIGNON
 - *SRVMAP
 - System value QALWOBJRST must be *ALL or (*ALWSYSSTT and *ALWPGMADP)

After installation, the following new libraries are on your server:

- ▶ QIDRGUI
- ▶ QIDRWCH

Installing Job Watcher

The GUI install process installs the server objects and programs on your iSeries server and the client code on your PC. The PTFs required for Job Watcher must be installed manually via whichever method you normally use to obtain PTFs for your iSeries. The PTFs required as of January 2005 were MF34003 and SI15800.

After the PTFs are loaded and applied on your iSeries server, perform the following steps:

1. Double-click the install image in Windows Explorer. It is a self-extracting .exe file.
2. Figure A-1 shows the self-extractor window. If desired, change the path where the install image is extracted on the PC and click **Unzip**. Wait a moment for the files to be extracted and the setup program to be launched.

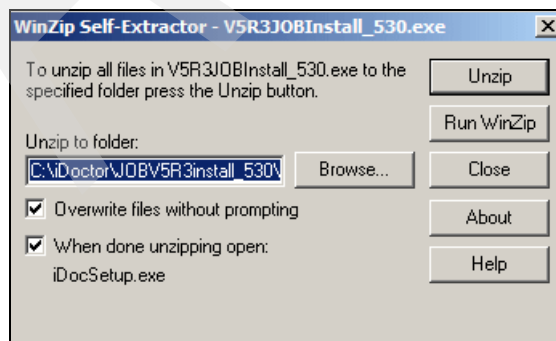


Figure A-1 WinZip Self Extractor window

3. Figure A-2 identifies the installed version of IBM iSeries Access for Windows as well as the version of iDoctor for iSeries client (if found). Click **Next**.

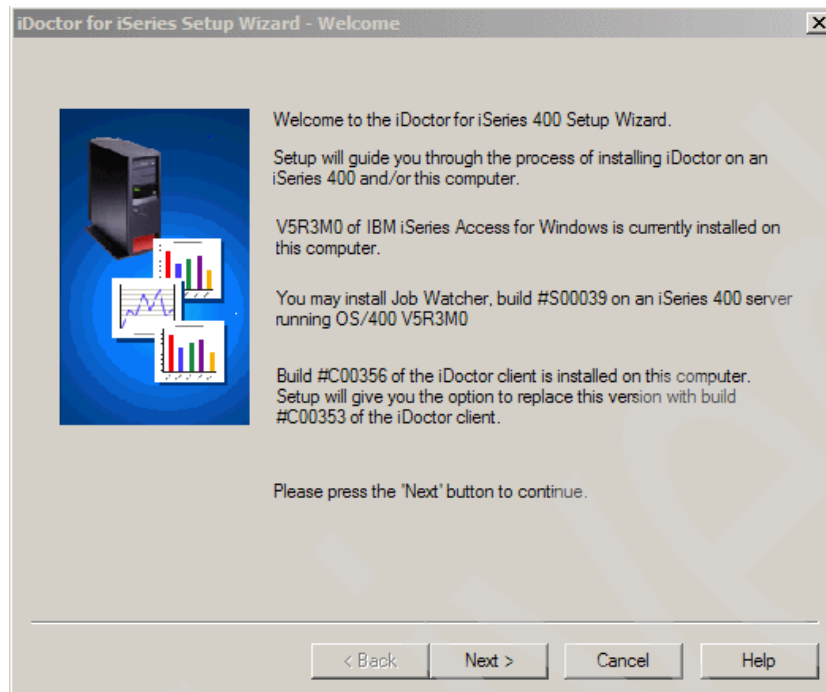


Figure A-2 iDoctor for iSeries Setup Wizard: Welcome

4. Figure A-3 indicates your acceptance of the license agreement. Click **Next** to agree and continue.

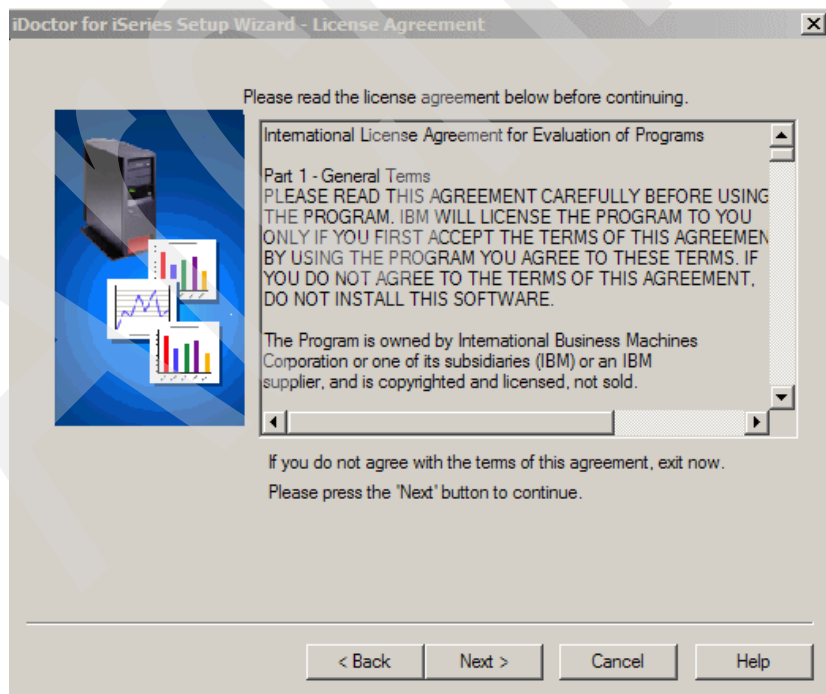


Figure A-3 iDoctor for iSeries Setup Wizard: License Agreement

5. Select the type of installation to perform. Figure A-4 enables you to choose whether to install the server side of Job Watcher, the client side of iDoctor, or both. Click **Next**.

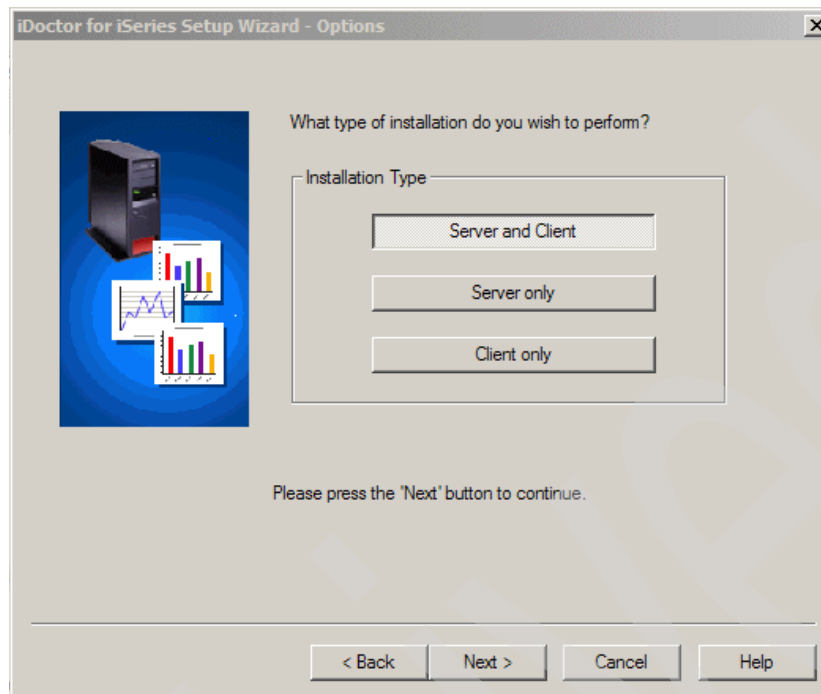


Figure A-4 iDoctor for iSeries Setup Wizard: Options

6. To install the server portion of Job Watcher, Figure A-5 asks for the connection information to use to access the server. The user profile must have the user class authority of *SECOFR and *ALLOBJ, *SECADM special authorities. Click **Next** to connect to the specified server and continue.

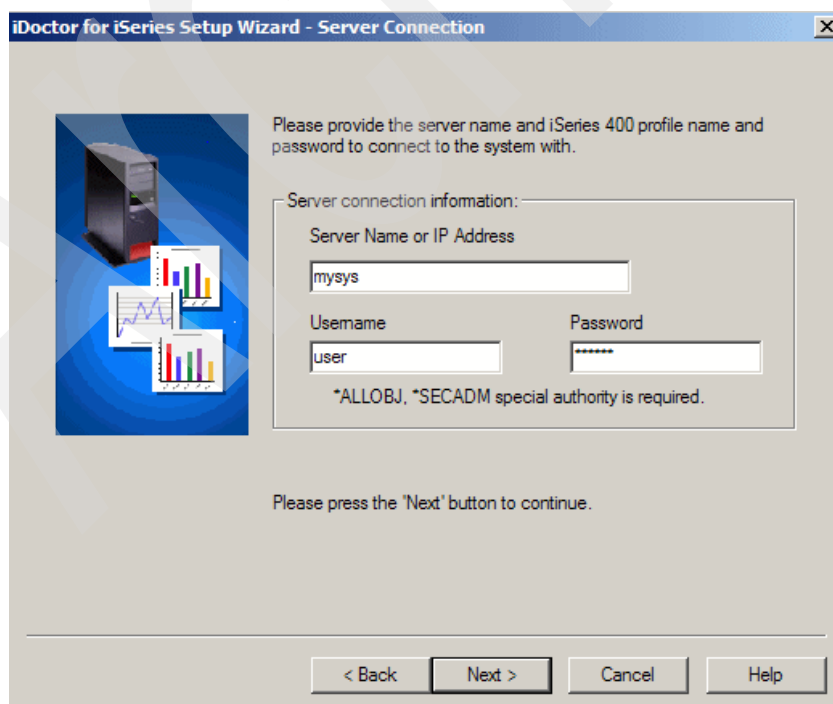
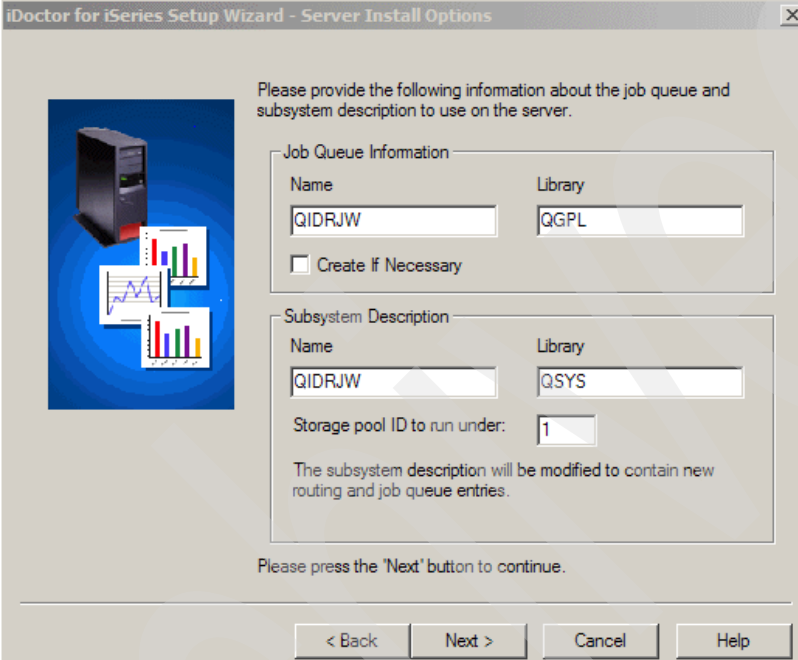


Figure A-5 iDoctor for iSeries Setup Wizard: Server Connection

7. To install the server portion of Job Watcher, specify a job queue name and a subsystem description that the server code may use when running Job Watcher (Figure A-6). If the subsystem does not already exist, the install program asks whether it should be created. This subsystem and job queue are used to run the Job Watcher watch jobs named QWCHJOB. Job Watcher does not have a limit to the number of watches that can be active at one time and the max active parameter value is set accordingly.

You must also indicate the storage pool ID that the Job Watcher jobs should run under. This parameter is required in order to add routing entries to the subsystem description.

Click **Next** to continue.



The screenshot shows a Windows-style dialog box titled "iDoctor for iSeries Setup Wizard - Server Install Options". On the left is a graphic of a server tower and charts. The main text says: "Please provide the following information about the job queue and subsystem description to use on the server." There are two sections: "Job Queue Information" and "Subsystem Description".

Job Queue Information	
Name	Library
QIDRJW	QGPL
<input type="checkbox"/> Create If Necessary	

Subsystem Description	
Name	Library
QIDRJW	QSYS
Storage pool ID to run under:	1
The subsystem description will be modified to contain new routing and job queue entries.	

At the bottom, it says "Please press the 'Next' button to continue." and has four buttons: "< Back", "Next >", "Cancel", and "Help".

Figure A-6 iDoctor for iSeries Setup Wizard: Server Install Options

8. Figure A-7 gives you the option to specify which type of FTP connection to use when performing the install. Only in unusual circumstances should anything other than the defaults be used. However, if installing over a VPN connection and Passive FTP does not work, try using Active FTP instead. Click **Next**.

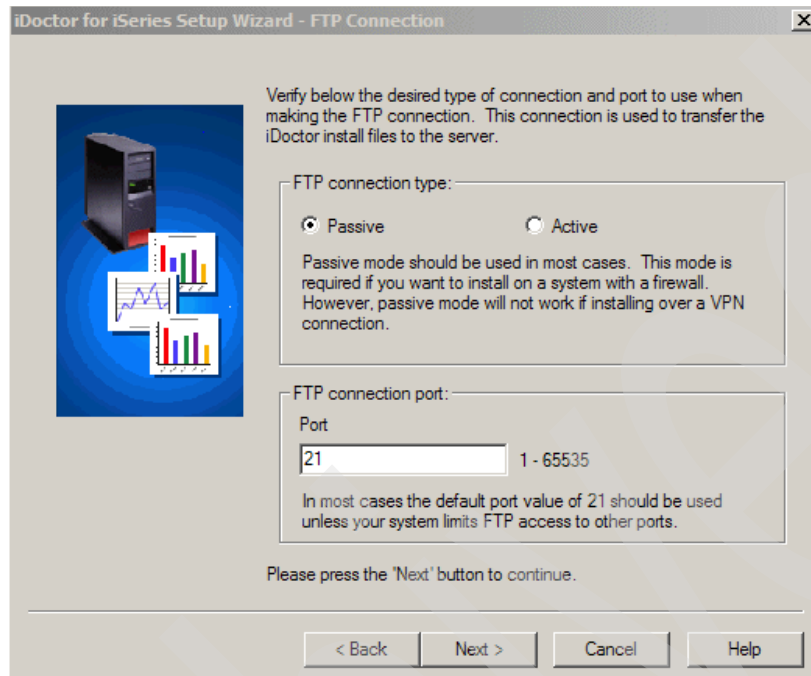


Figure A-7 iDoctor for iSeries Setup Wizard: FTP Connection

9. If an access code is required, Figure A-8 appears. Enter the access code (optional) and click **Next** to continue.

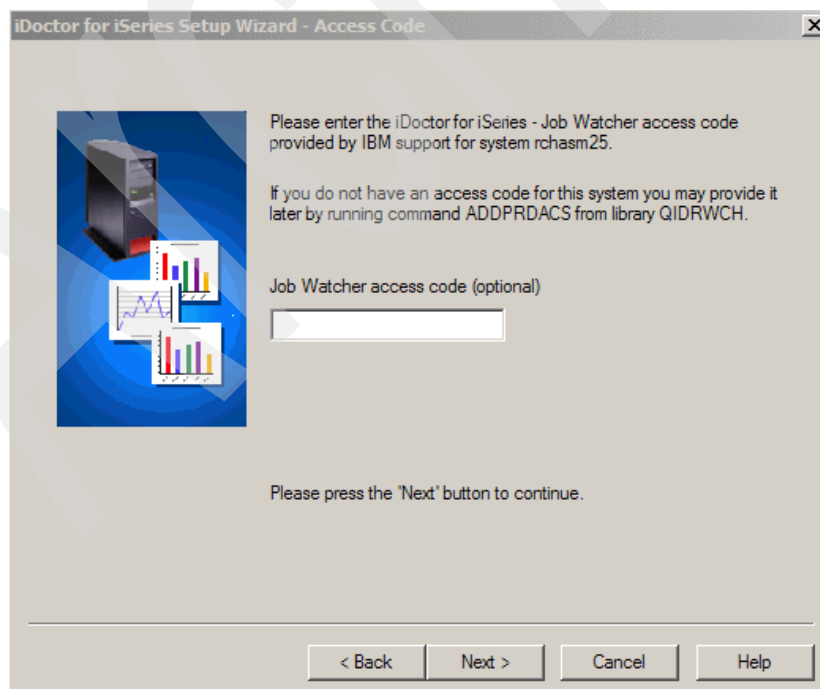


Figure A-8 iDoctor for iSeries Setup Wizard: Access Code

10. A summary of your selections appears. This window also has options for deleting the temporary libraries used during the installation. During the install process these libraries hold the restored contents of the save files sent to the server. Keep these only if you have problems with the install. They can be used by IBM service for problem determination.

Click **Finish** to copy all of the files and run the commands that are needed to install the this portion of Job Watcher. The server portion of the installation may take a few minutes.

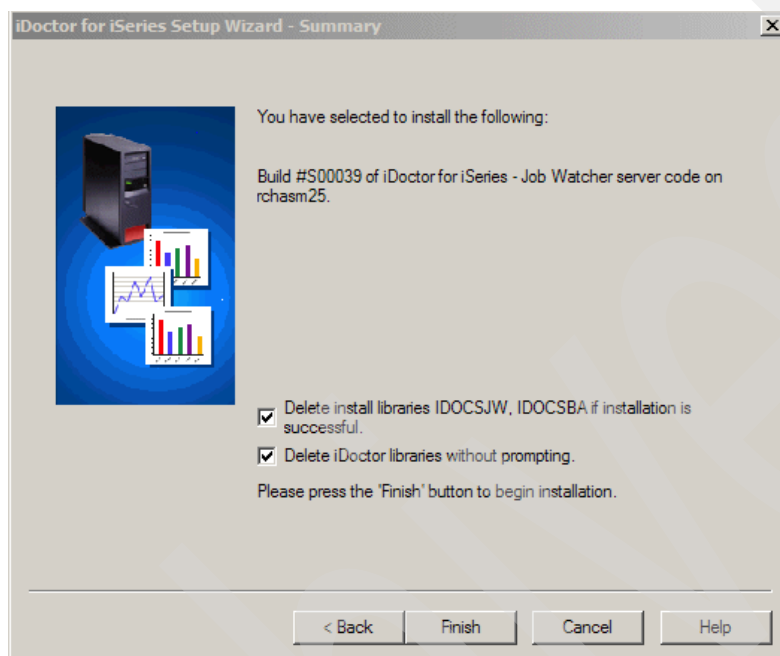


Figure A-9 iDoctor for iSeries Setup Wizard: Summary

After the install completes, the setup log file appears. If any errors occur, send this file to idoctor@us.ibm.com for assistance.

Uninstalling Job Watcher

This section has the uninstall process for Job Watcher on both the server and the client side.

Note: This section does not apply to the PTDV component.

Server side

To uninstall an iDoctor for iSeries component, remove the libraries created during the installation. Table A-1 shows the libraries installed on the server by component.

Table A-1 Install libraries on the server by component

	Heap Analyzer	PEX Analyzer	Job Watcher
Libraries	QIDRGUI, QIDRHAJ	QIDRGUI, QIDRPA	QIDRGUI, QIDRWCH

Client side

To remove iDoctor for iSeries from your PC, select the uninstall program from the Start Menu. Select **Start** → **Programs** → **iDoctor for iSeries** → **Uninstall iDoctor for iSeries**.

Archived

Database files created by Job Watcher

This appendix describes all of the database files that can be produced by the Job Watcher collection function using either the Windows operating system graphical interface or the WCHJOB command in library QIDRWCH.

We recommend that you review this appendix before writing your own queries over these files to get the most out of this data.

This appendix covers the files created in the user's library specified for a Job Watcher collection:

- ▶ Files created by every Job Watcher collection, irrespective of the data collection options you choose
- ▶ Files created only when specific Job Watcher collection options are used
- ▶ Files created only when you run queries or graphs

Job Watcher terminology

To understand Job Watcher graphs and reports, you must be familiar with some terminology. This terminology topic is replicated from Chapter 2, “Overview of job waits and iDoctor for iSeries Job Watcher” on page 21, for ease of reference when examining the Job Watcher file and field within file descriptions in this appendix.

► Thread

A thread is a separate and unique flow of control within a job. A thread runs a procedure asynchronously to other threads running within the job. The term thread is an above-TIMI (Technology Independent Machine Interface) term; the LIC actually uses the term *task count*.

A single job can have one or many threads. It always has one single primary thread (also called an initial thread) and may have zero to many secondary threads.

The work performed by a job is the sum of work performed by all threads within that job plus the work that is done for that job by other system tasks or jobs.

A LIC task does not have threads.

► Task count

This term is used within the LIC and is discussed here because it may appear in detailed Job Watcher information. At the operating system level it is analogous to a thread number, but does not have the same value as the thread number at the microcode level. The microcode task dispatcher manages tasks using the task count value.

A task count is a unique numeric identifier assigned by the microcode to every dispatchable unit of work. Every job thread and LIC task has a unique task count value, assigned in ascending sequence, starting when the system or partition is IPLed. (also termed *started* or *activated*). A task count value is unique within a partition while that partition is active.

A task count is associated with a TDE, which is a control block representing an individual task or individual thread within a job. Within a job, there is one task count that represents the primary thread, and zero to many unique, different task counts that represent each of any secondary threads.

Note that at the microcode level each task count has a run priority that was either established at the operating system level (such as via the job's run priority parameter) or according to microcode implementation requirements.

The task count is used in several of the queries we use on Job Watcher collected data throughout this book. It is used, for example, to get a thread's call stack information.

In some other iSeries documentation and performance tools, the task count is also referred to as *task ID*.

► TDE

This stands for Task Dispatcher Element, which is a LIC control block that anchors every dispatchable unit (thread or task) to be run in the system. The tasking component of the iSeries system uses the TDE when dispatching (assigning work to run on a processor) work. Note that many of the performance metrics are scoped to a thread or task in a TDE and harvested by Job Watcher from the TDE.

For this book, assume that a TDE is associated with a single thread of a job or a single LIC task.

The TDE contains both the task count and the accounting data necessary for running the thread or task in priority among other threads or tasks and for basic performance analysis.

While not showing the TDE values, operating system thread values can be seen on several operating system command screens, such as the Work with Active Jobs - job details - Work with Threads option display a “thread number.”

Task count values are shown only on low-level tools, such as Job Watcher output.

The Performance Tools for iSeries Work with System Activity command shows active operating system thread numbers and LIC tasks.

The thread or job information that can be shown using these WRKACTJOB/WRKSYSACT command interfaces includes information stored within the TDE.

- Snapshot

A Job Watcher collection sample of the selected job threads and tasks.

- Interval

The delta time between snapshots. An interval number is stored in Job Watcher files for chronological sequencing. The interval size is based on the data collection interval that is selected.

- Eye catcher

An eye catcher is a four-character (three meaningful characters followed by a blank character) short description acronym that indicates an individual type of low-level wait status of a job, thread, or task within the system. It may or may not indicate why the job is not progressing.

For example, an eye catcher of **Rex** represents a wait for a request for an **exclusive** seize and has an ENUM of 101.

- ENUM

An ENUM is how the system internally identifies an individual type of low-level wait. A single ENUM has only one eye catcher; however, a single eye catcher may be associated with several different ENUMs.

An ENUM is also known as a *block identifier* (where block is used as in “blocking the path.”).

See 2.8, “Waiting point groupings (wait buckets)” on page 29 for a list of all possible waits (eye catchers) and their corresponding ENUMs and bucket mappings for V5R3.

- Bucket number

A bucket number is also known as a *bucket identifier*, *wait bucket*, *block bucket*, or *queueing bucket identifier*.

There are 32 separate bucket identifiers for every job/thread and task on the system. Each bucket identifier has two parts:

- Counts
- Durations

At V5R3, there are 199 different ENUMs, each of which represents an individual low-level wait. These 199 ENUMs are mapped into the 32 buckets; hence the counts and durations of several related ENUMs are mapped into the same bucket. This grouping into buckets is done to minimize overhead of extra information being stored in the TDE.

For example, the description of queueing bucket 15 is *seize contention*, and 20 different ENUMs are mapped to this bucket identifier. Two of the eye catchers associated with these ENUMs show **Rex** and **Rsh**, which identify the individual types of seize requests as *exclusive seize* and *shared seize*, respectively.

Files created by every Job Watcher collection

The eight files in Table B-1 are always created for every GUI-initiated collection, irrespective of which data collection options you selected. These comprise the base file set of every collection.

The *Job Watch name* you specify for a GUI collection becomes the member name in each of these files except the QAIDRJWRI file.

Table B-1 Files in every Job Watcher collection

File name	Description	Comments
QAIDRJWCPU	CPU utilization statistics	CPU utilization by interval
QAIDRJWRI	Run information	One member named QAIDRJWRI for <i>all</i> collections. Includes collection end status and collection member name (Job Watch name).
QAIDRJWDFN	Job watch definitions	Job watch definitions created by the GUI, including the full QIDRWCH/WCHJOB command used to collect the data. ^a
QAPYJWBKT	Job wait bucket information	The mapping of ENUMs to bucket numbers at the time of data collection. Includes the eye catcher for each ENUM.
QAPYJWINTI	Basic interval information	Counts of all threads and tasks in the system and in your selection, and all active threads and tasks in your selection, by interval.
QAPYJWJVM	Java JVM scoped data	
QAPYJWJVTH	Java TDE data	
QAPYJWPRC	Main process scoped data	
QAPYJWRUNI	Basic collection and system information	Basic information that does not include collection member name (Job Watch name) or collection end status.
QAPYJWSTS	Status information	One record per interval for every collected TDE existing at sample interval end. The status and last wait type that occurred for the TDE before interval end are provided. If the TDE used CPU in that interval, then the status will be active. If no CPU is used then the status will be idle. If the status is active, then the last current wait is already added into the appropriate wait bucket in the QAPYJWTDE file by the JW Collector. If the status is idle, the wait is still in progress and there is no record in the QAPYJWTDE file for this TDE. The status of selected TDEs at the end of each interval snapshot time, including the ENUM of its longest wait during that interval.
QAPYJWTDE	Main TDE scoped information	The master file for Job Watcher collections. This file contains fields mostly, but not entirely, common to both threads and LIC tasks. Some fields, such as THREADID (and other fields associated with a thread) are only applicable to operating system level jobs and threads. No fields are applicable only to LIC tasks. This file contains a record for every task count that used CPU during an interval. Each record contains detailed information about what state a selected thread was in as well as counts and times of waits, detailed I/Os, dispatched CPU, CPU queueing, and more.
a. If you collect data using the QIDRWCH/WCHJOB command and not the GUI, then this file is not created.		

Files created only when call stack information is specified in a Job Watcher collection

The two files in Table B-2 are created only if your collection included call stack information.

Table B-2 Files created only when call stack information is collected

File name	Description
QAPYJWPROC	Procedure information
QAYPYJWSTK	Call stack information

Files created only when SQL information is specified in a Job Watcher collection

The four files in Table B-3 are created only if your collection included SQL information. Review the comments column to see which collection configuration options must be used for each of these four files to be created.

Table B-3 Files created only when SQL information is collected

File name	Description	Comments
QAPYJWSQL	SQL statement information	Always created when you collect any type of SQL information. This file does not contain SQL statements with host variables resolved. ^a
QAYPYJWSQLH	SQL host variable information	Always created when you collect any type of SQL information. This file contains the host variables for SQL in file QAPYJWSQL. ^a
QAPYJWSQLO	SQL open cursor list (OCL) information	Created only when you collect last executed SQL statements, host variables, prepared statement arrays and open cursors. ^b
QAPYJWSQLP	SQL prepared statement area (PSA) information	Created only when you collect last executed SQL statements, host variables, prepared statement arrays and open cursors. ^b
<p>a. SQL statements with host variable information resolved can be viewed only by using the Job Watcher GUI. There are no green screen commands that enable this resolution.</p> <p>b. When you collect last-executed SQL statements, that statement could have completed 2 hours ago and taken 5 minutes to run. Be careful what assumptions you make if you use the last executed SQL statement collection option.</p>		

Files created only when activation group information is specified in a Job Watcher collection

The three files in Table B-4 are created only if your Job Watcher collection included activation group information. Review the comments column to see which collection configuration options must be used for each of these files to be created.

Table B-4 Files created only when activation group information is collected

File name	Description	Comment
QAPYJWAIGP	Activation group information	Created only if you collect activation group details. If you collect activation group counter information then some of this data is also summarized in the base collection file QAPYJWPRC.
QAYPYJWAIHP	Heap information for an activation group	Created only if you collect activation group details. If you collect only activation group counter information, this data is not in the base collection file QAPYJWPRC.
QAPYJWAIPA	Program activation group information	Created only if you collect activation group details. If you collect only activation group counter information, some of this data is summarized in the base collection file QAPYJWPRC.

Files created only when sockets information is specified in a Job Watcher collection

The two files in Table B-5 are created only if your collection included sockets information.

Table B-5 Files created only when sockets information is collected

File name	Description
QAPYJWSKJB	Jobs associated with sockets
QAYPYJWSKTC	Sockets and TCP endpoint information

Description of ENUM to queueing bucket identifier mappings

Example B-1 is from a query that joins the ENUM file QIDRJWENM in the Job Watcher QIDRWCH library to the queueing bucket file QAPYJWBKT in the collection library. The QIDRJWENM file is included as part of Job Watcher installation on the iSeries system.

The example shows the SQL used to create this query. The description headings for bucket and ENUM are shown abbreviated for the sake of readability. You must issue an override file command (OVRDBF) for the QAPYJWBKT file to the Job Watcher collection member name before this SQL statement is run.

Example: B-1 SQL example showing ENUM-to-queueing-bucket mappings

```
SELECT dec(a.enum,3,0) as ENUM, dec(bucketnum,3,0) as BKTNUM,  
eye, substr(enumdesc,1,68) as enumdesc, substr(bucketdesc,12,31) as  
bucketdesc  
FROM qidrwch/qaidrjwenm a, redbookdh/qapyjwbkt b  
WHERE a.enum = b.enum
```


and b.eye <> 'XXX'
order by enum

The output from this SQL provides the descriptions for the individual waits (eye catcher for ENUM) and the associated queueing bucket identifier (number). Remember that there may be several different ENUMs mapped to a single queueing bucket number.

Description of ENUM to queueing bucket mappings

Example B-2 provides the queueing bucket identifiers for all ENUMs (individual low-level waits) in use in Job Watcher at V5R3, as well as the associated three-character eye catcher.

Example: B-2 ENUM and queueing bucket descriptions (shown on next several pages)

ENUM	BKTNUM	Eye catcher	ENUMDESC	BUCKETDESC
1	18	QCo	QUCOUNTER, NOT OTHERWISE IDENTIFIED	Other waits
2	14	QGa	QUGATE, NOT OTHERWISE IDENTIFIED	MACHINE LEVEL GATE SERIALIZATIO
3	14	QTG	QUTRYGATE, NOT OTHERWISE IDENTIFIED	MACHINE LEVEL GATE SERIALIZATIO
4	18	QTB	QUSINGLETASKBLOCKER, NOT OTHERWISE IDENTIFIED	Other waits
5	18	QUW	QUUNBLOCKWHENDONE, NOT OTHERWISE IDENTIFIED	Other waits
6	18	QQu	QUQUEUE, NOT OTHERWISE IDENTIFIED	Other waits
7	18	QTQ	QUTREEQUEUE, NOT OTHERWISE IDENTIFIED	Other waits
8	32	QRP	QURESSTACKMSGPOOL, NOT OTHERWISE IDENTIFIED	ABNORMAL CONTENTION
9	18	QPo	QUPOOL, NOT OTHERWISE IDENTIFIED	Other waits
10	18	QMP	QUMESSAGEPOOL, NOT OTHERWISE IDENTIFIED	Other waits
11	18	QMP	QUSIMPLEMSGPOOL, NOT OTHERWISE IDENTIFIED	Other waits
12	18	QSP	QUSTACKLESSMSGPOOL, NOT OTHERWISE IDENTIFIED	Other waits
13	18	QSC	QUSTATECOUNTER, NOT OTHERWISE IDENTIFIED	Other waits
14	32	QWL	QUWAITLIST, WAITING FOR ACCESS TO A WAIT LIST	ABNORMAL CONTENTION
15	12	QMG	QUMUTEXGATE, NOT OTHERWISE IDENTIFIED	MUTEX/SEMAPHORE CONTENTION
16	12	QSm	QUSEMAPHORE, NOT OTHERWISE IDENTIFIED	MUTEX/SEMAPHORE CONTENTION
17	18	QSB	QUSYSTEMBLOCKER, NOT OTHERWISE IDENTIFIED	Other waits
40	32	QGb	QUGATEB, ABNORMAL QUGATE CONTENTION, FIRST RETRY	ABNORMAL CONTENTION
41	32	QGc	QUGATEC, ABNORMAL QUGATE CONTENTION, SECOND RETRY	ABNORMAL CONTENTION
42	32	QGd	QUGATED, ABNORMAL QUGATE CONTENTION, THIRD RETRY	ABNORMAL CONTENTION
49	11	QRR	QURESSTACKMSGPOOL, ABNORMAL DASD OP START CONTENTION	DASD OP-START CONTENTION
50	13	JBo	JOURNAL BUNDLE OWNER WAIT FOR DASD COMPLETION	JOURNAL SERIALIZATION
51	13	JBw	JOURNAL BUNDLE WAIT FOR DASD COMPLETION	JOURNAL SERIALIZATION
60	10	DSM	DASD MANAGEMENT OPS: FIND COMPRESSION GROUP	DASD (OTHER READS OR WRITES)
61	10	DSM	DASD MANAGEMENT OPS: DEALLOCATE COMPRESS GROUP	DASD (OTHER READS OR WRITES)
62	10	DSM	DASD MANAGEMENT OPS: READ COMPRESSION DIRECTORY	DASD (OTHER READS OR WRITES)
63	10	DSM	DASD MANAGEMENT OPS: WRITE COMPRESSION DIRECTORY	DASD (OTHER READS OR WRITES)
64	10	DSM	DASD MANAGEMENT OPS: INIT COMPRESSION START REORG	DASD (OTHER READS OR WRITES)
65	10	DSM	DASD MANAGEMENT OPS: MIRROR READ SYNC	DASD (OTHER READS OR WRITES)
66	10	DSM	DASD MANAGEMENT OPS: MIRROR REASSIGN SYNC	DASD (OTHER READS OR WRITES)
67	10	DSM	DASD MANAGEMENT OPS: MIRROR WRITE VERIFY SYNC	DASD (OTHER READS OR WRITES)
68	10	DSM	DASD MANAGEMENT OPS: READ	DASD (OTHER READS OR WRITES)
69	10	DSM	DASD MANAGEMENT OPS: READ DIAG	DASD (OTHER READS OR WRITES)
70	10	DSM	DASD MANAGEMENT OPS: VERIFY	DASD (OTHER READS OR WRITES)
71	10	DSM	DASD MANAGEMENT OPS: VERIFY DIAG	DASD (OTHER READS OR WRITES)
72	10	DSM	DASD MANAGEMENT OPS: WRITE	DASD (OTHER READS OR WRITES)
73	10	DSM	DASD MANAGEMENT OPS: WRITE DIAG	DASD (OTHER READS OR WRITES)
74	10	DSM	DASD MANAGEMENT OPS: WRITE VERIFY	DASD (OTHER READS OR WRITES)
75	10	DSM	DASD MANAGEMENT OPS: WRITE VERIFY DIAG	DASD (OTHER READS OR WRITES)
76	10	DSM	DASD MANAGEMENT OPS: REASSIGN	DASD (OTHER READS OR WRITES)
77	10	DSM	DASD MANAGEMENT OPS: REASSIGN DIAG	DASD (OTHER READS OR WRITES)
78	10	DSM	DASD MANAGEMENT OPS: ALLOCATE	DASD (OTHER READS OR WRITES)
79	10	DSM	DASD MANAGEMENT OPS: ALLOCATE DIAG	DASD (OTHER READS OR WRITES)
80	10	DSM	DASD MANAGEMENT OPS: DEALLOCATE	DASD (OTHER READS OR WRITES)
81	10	DSM	DASD MANAGEMENT OPS: DEALLOCATE DIAG	DASD (OTHER READS OR WRITES)
82	10	DSM	DASD MANAGEMENT OPS: ENABLE AUTO ALLOCATE	DASD (OTHER READS OR WRITES)
83	10	DSM	DASD MANAGEMENT OPS: DISABLE AUTO ALLOCATE	DASD (OTHER READS OR WRITES)
84	10	DSM	DASD MANAGEMENT OPS: QUERY COMPRESSION METRICS	DASD (OTHER READS OR WRITES)

85	10	DSM	DASD MANAGEMENT OPS: QUERY COMPRESSION METRICS DIAG	DASD (OTHER READS OR WRITES)
86	10	DSM	DASD MANAGEMENT OPS: COMPRESSION SCAN READ	DASD (OTHER READS OR WRITES)
87	10	DSM	DASD MANAGEMENT OPS: COMPRESSION SCAN READ DIAG	DASD (OTHER READS OR WRITES)
88	10	DSM	DASD MANAGEMENT OPS: COMPRESSION DISCARD TEMP DATA	DASD (OTHER READS OR WRITES)
89	10	DSM	DASD MANAGEMENT OPS: COMPRESSION DISCARD TEMP DATA DIAG	DASD (OTHER READS OR WRITES)
100	15	Rex	SEIZE: EXCLUSIVE	SEIZE CONTENTION
101	15	Rex	SEIZE: LONG RUNNING EXCLUSIVE	SEIZE CONTENTION
102	15	Rsh	SEIZE: SHARED	SEIZE CONTENTION
103	15	Rix	SEIZE: INTENT EXCLUSIVE	SEIZE CONTENTION
104	15	Ris	SEIZE: INTENT SHARED	SEIZE CONTENTION
105	15	Rfa	SEIZE: FLUSH ALL	SEIZE CONTENTION
106	15	Rdx	SEIZE: DATABASE EXCLUSIVE	SEIZE CONTENTION
107	15	Rii	SEIZE: INTERNAL INTENT EXCLUSIVE	SEIZE CONTENTION
108	15	Rot	SEIZE: OTHER	SEIZE CONTENTION
109	15	Rlk	SEIZE: LOCK CONFLICT	SEIZE CONTENTION
110	16	RDr	DB RECORD LOCK: READ	DATABASE RECORD LOCK CONTENTION
111	16	RDu	DB RECORD LOCK: UPDATE	DATABASE RECORD LOCK CONTENTION
112	15	RXX	SEIZE/LOCK IMPOSSIBLE	SEIZE CONTENTION
113	17	RIr	LOCK: SHARED READ	OBJECT LOCK CONTENTION
114	17	RIo	LOCK: SHARED READ ONLY	OBJECT LOCK CONTENTION
115	17	RIu	LOCK: SHARED UPDATE	OBJECT LOCK CONTENTION
116	17	RIa	LOCK: EXCLUSIVE ALLOW READ	OBJECT LOCK CONTENTION
117	17	RIe	LOCK: EXCLUSIVE NO READ	OBJECT LOCK CONTENTION
118	17	RMr	LOCK: SEIZE CONFLICT, EXCLUSIVE	OBJECT LOCK CONTENTION
119	17	RMo	LOCK: SEIZE CONFLICT, SHARED	OBJECT LOCK CONTENTION
120	17	RMu	LOCK: SEIZE CONFLICT, INTENT EXCLUSIVE	OBJECT LOCK CONTENTION
121	17	RMa	LOCK: SEIZE CONFLICT, INTENT SHARED	OBJECT LOCK CONTENTION
122	17	RMe	LOCK: SEIZE CONFLICT, INTERNAL INTENT EXCLUSIVE	OBJECT LOCK CONTENTION
123	16	RDw	DB RECORD LOCK: WEAK	DATABASE RECORD LOCK CONTENTION
124	17	RMm	LOCK: MATERIALIZE	OBJECT LOCK CONTENTION
125	15	Rsp	SEIZE: OFF-LINE IASP	SEIZE CONTENTION
126	15	Rra	SEIZE: RELEASE ALL	SEIZE CONTENTION
127	15	Rrs	SEIZE: RELEASE	SEIZE CONTENTION
128	17	Rdo	LOCK: DESTROY OBJECT	OBJECT LOCK CONTENTION
129	17	Rdp	LOCK: DESTROY PROCESS	OBJECT LOCK CONTENTION
130	17	Rdt	LOCK: DESTROY THREAD	OBJECT LOCK CONTENTION
131	17	Rdx	LOCK: DESTROY TRXM	OBJECT LOCK CONTENTION
132	17	Rar	LOCK: ASYNC RETRY	OBJECT LOCK CONTENTION
133	15	Rss	SEIZE/LOCK: INTERNAL SERVICE TOOLS HASH CLASS GATE	SEIZE CONTENTION
134	16	Rxf	DB RECORD LOCK: TRANSFER	DATABASE RECORD LOCK CONTENTION
135	15	Rmf	SEIZE: MONITORED FREE	SEIZE CONTENTION
136	16	Rck	DB RECORD LOCK: CHECK	DATABASE RECORD LOCK CONTENTION
137	17	Rtr	LOCK: TRACE	OBJECT LOCK CONTENTION
138	17	Rul	LOCK: UNLOCK	OBJECT LOCK CONTENTION
139	16	Rcx	DB RECORD LOCK: CONFLICT EXIT	DATABASE RECORD LOCK CONTENTION
140	17	Rlc	LOCK: LOCK COUNT	OBJECT LOCK CONTENTION
141	15	Rcu	SEIZE: CLEANUP	SEIZE CONTENTION
142	17	Rpi	LOCK: PROCESS INTERRUPT	OBJECT LOCK CONTENTION
143	15	Rsv	SEIZE/LOCK: SERVICE	SEIZE CONTENTION
145	7	ASM	DASD SPACE MANAGER: CONCURRENCY CONTENTION	DASD SPACE USAGE CONTENTION
146	7	ASM	DASD SPACE MANAGER: ASP FREE SPACE DIRECTORY	DASD SPACE USAGE CONTENTION
147	7	ASM	DASD SPACE MANAGER: RR FREE SPACE LOCK	DASD SPACE USAGE CONTENTION
148	7	ASM	DASD SPACE MANAGER: GP FREE SPACE LOCK	DASD SPACE USAGE CONTENTION
149	7	ASM	DASD SPACE MANAGER: PERMANENT DIRECTORY LOCK	DASD SPACE USAGE CONTENTION
150	10	STv	MAINSTORE/LOGICAL-DASD-IO: SAR NOT SET	DASD (OTHER READS OR WRITES)
151	10	SRv	MAINSTORE/LOGICAL-DASD-IO: REMOVE	DASD (OTHER READS OR WRITES)
152	10	SRP	MAINSTORE/LOGICAL-DASD-IO: REMOVE IO PENDING	DASD (OTHER READS OR WRITES)
153	10	SCl	MAINSTORE/LOGICAL-DASD-IO: CLEAR	DASD (OTHER READS OR WRITES)
154	10	SCP	MAINSTORE/LOGICAL-DASD-IO: CLEAR IO PENDING	DASD (OTHER READS OR WRITES)
155	19	GCP	MAINSTORE/LOGICAL-DASD-IO: CLEAR PAGE OUT WAIT	MAIN STORAGE POOL OVERCOMMITMEN
156	10	SUp	MAINSTORE/LOGICAL-DASD-IO: UNPIN	DASD (OTHER READS OR WRITES)
157	10	SUP	MAINSTORE/LOGICAL-DASD-IO: UNPIN IO PENDING	DASD (OTHER READS OR WRITES)
158	6	SRd	MAINSTORE/LOGICAL-DASD-IO: DASD READ	DASD non fault reads
159	6	SRQ	MAINSTORE/LOGICAL-DASD-IO: DASD READ IO PENDING	DASD non fault reads
160	19	GRQ	MAINSTORE/LOGICAL-DASD-IO: DASD READ PAGE OUT WAIT	MAIN STORAGE POOL OVERCOMMITMEN
161	5	SFt	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT	DASD (PAGE FAULTS)
162	5	SFP	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT IO PENDING	DASD (PAGE FAULTS)
163	19	GFP	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT PAGE OUT WAIT	MAIN STORAGE POOL OVERCOMMITMEN
164	5	GRf	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT	DASD (PAGE FAULTS)
165	5	SRR	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT IO PENDING	DASD (PAGE FAULTS)

166	19	GRR	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT PAGE OUT WAIT	MAIN STORAGE POOL OVERCOMMITMEN
167	9	SWt	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE	DASD WRITES
168	9	SWP	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE IO PENDING	DASD WRITES
169	19	GWP	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE PAGE OUT WAIT	MAIN STORAGE POOL OVERCOMMITMEN
170	9	SWp	MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WRITE	DASD WRITES
171	9	GPg	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE	DASD WRITES
172	9	GPP	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE IO PENDING	DASD WRITES
173	19	SPw	MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WAIT, POOL OVERCOMMITMENT	MAIN STORAGE POOL OVERCOMMITMEN
174	9	GTA	MAINSTORE/LOGICAL-DASD-IO: GENERIC ASYNC IO TRACKER WAIT	DASD WRITES
175	9	GTS	MAINSTORE/LOGICAL-DASD-IO: GENERIC SINGLE TASK BLOCKER WAIT	DASD WRITES
176	9	GTT	MAINSTORE/LOGICAL-DASD-IO: GENERIC TIMED TASK BLOCKER	DASD WRITES
177	10	SMP	MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION	DASD (OTHER READS OR WRITES)
178	10	SMC	MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION CHANGE	DASD (OTHER READS OR WRITES)
180	7	ASM	DASD SPACE MANAGER: TEMPORARY DIRECTORY LOCK	DASD SPACE USAGE CONTENTION
181	7	ASM	DASD SPACE MANAGER: PERSISTENT STORAGE LOCK	DASD SPACE USAGE CONTENTION
182	7	ASM	DASD SPACE MANAGER: STATIC DIRECTORY LOCK	DASD SPACE USAGE CONTENTION
183	7	ASM	VIRTUAL ADDRESS MANAGER: BIG SEGMENT ID LOCK	DASD SPACE USAGE CONTENTION
184	7	ASM	VIRTUAL ADDRESS MANAGER: LITTLE SEGMENT ID LOCK	DASD SPACE USAGE CONTENTION
185	7	ASM	DASD SPACE MANAGER: IASP LOCK	DASD SPACE USAGE CONTENTION
186	7	ASM	DASD SPACE MANAGER: MOVE CHAIN	DASD SPACE USAGE CONTENTION
187	7	ASM	DASD SPACE MANAGER: HYPERSPACE LOCK	DASD SPACE USAGE CONTENTION
188	7	ASM	DASD SPACE MANAGER: NON PERSISTENT DATA LOCK	DASD SPACE USAGE CONTENTION
189	7	ASM	VIRTUAL ADDRESS MANAGER: TEMPORARY SEGMENT ID RANGE MAPPER LOCK	DASD SPACE USAGE CONTENTION
190	7	ASM	VIRTUAL ADDRESS MANAGER: PERMANENT SEGMENT ID RANGE MAPPER LOCK	DASD SPACE USAGE CONTENTION
191	7	ASM	VIRTUAL ADDRESS MANAGER: IASP SEGMENT ID RANGE MAPPER LOCK	DASD SPACE USAGE CONTENTION
200	20	JUW	JAVA: USER WAIT	JAVA USER (INCLUDING LOCKS)
201	20	JSL	JAVA: USER SLEEP	JAVA USER (INCLUDING LOCKS)
202	22	JWC	JAVA: WAIT FOR COUNT	JAVA (OTHER)
203	20	JSU	JAVA: SUSPEND WAIT	JAVA USER (INCLUDING LOCKS)
204	22	JEa	JAVA: END ALL THREADS	JAVA (OTHER)
205	22	JDE	JAVA: DESTROY WAIT	JAVA (OTHER)
206	22	JSD	JAVA: SHUTDOWN	JAVA (OTHER)
207	22	JCL	JAVA: CLASS LOAD WAIT	JAVA (OTHER)
208	22	JSL	JAVA: SIMPLE LOCK	JAVA (OTHER)
209	20	JOL	JAVA: OBJECT LOCK	JAVA USER (INCLUDING LOCKS)
210	23	STA	COMM/SOCKETS: SHORT WAIT FOR ACCEPT	SOCKET ACCEPTS
211	23	LTA	COMM/SOCKETS: LONG WAIT FOR ACCEPT	SOCKET ACCEPTS
212	24	STS	COMM/SOCKETS: SHORT WAIT FOR TCP SEND	SOCKET TRANSMITS
213	24	LTS	COMM/SOCKETS: LONG WAIT FOR TCP SEND	SOCKET TRANSMITS
214	25	STR	COMM/SOCKETS: SHORT WAIT FOR TCP RECEIVE	SOCKET RECEIVES
215	25	LTR	COMM/SOCKETS: LONG WAIT FOR TCP RECEIVE	SOCKET RECEIVES
216	24	SUS	COMM/SOCKETS: SHORT WAIT FOR UDP SEND	SOCKET TRANSMITS
217	24	LUS	COMM/SOCKETS: LONG WAIT FOR UDP SEND	SOCKET TRANSMITS
218	25	SUR	COMM/SOCKETS: SHORT WAIT FOR UDP RECEIVE	SOCKET RECEIVES
219	25	LUR	COMM/SOCKETS: LONG WAIT FOR UDP RECEIVE	SOCKET RECEIVES
220	26	SAS	COMM/SOCKETS: SHORT WAIT FOR IO COMPLETION	SOCKET (OTHER)
221	26	LAS	COMM/SOCKETS: LONG WAIT FOR IO COMPLETION	SOCKET (OTHER)
222	26	SSW	COMM/SOCKETS: SELECT SHORT WAIT	SOCKET (OTHER)
223	26	SLW	COMM/SOCKETS: SELECT LONG WAIT	SOCKET (OTHER)
240	18	RCA	LIC CHAIN FUNCTIONS: SMART CHAIN ACCESS	Other waits
241	18	RCI	LIC CHAIN FUNCTIONS: SMART CHAIN ITERATOR	Other waits
242	18	RCM	LIC CHAIN FUNCTIONS: CHAIN MUTATOR	Other waits
243	18	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 1	Other waits
244	18	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 2	Other waits
245	18	RCE	LIC CHAIN FUNCTIONS: CHAIN ACCESS EXTENDED	Other waits
250	28	PRL	IFS/PIPE: FILE TABLE ENTRY EXCLUSIVE LOCK	IFS (OTHER)
251	28	PRC	IFS/PIPE: LIC REFERENCE COUNT	IFS (OTHER)
252	27	PPC	IFS/PIPE: MAIN PIPE COUNT	IFS PIPE
253	27	PRP	IFS/PIPE: READ END OF PIPE	IFS PIPE
254	27	PWP	IFS/PIPE: WRITE END OF PIPE	IFS PIPE
255	27	PRW	IFS/PIPE: PIPE READ WAITERS	IFS PIPE
256	27	PWW	IFS/PIPE: PIPE WRITE WAITERS	IFS PIPE
260	13	EFJ	EPFS: WAIT FOR OS TO FINISH APPLY JOURNAL CHANGES	JOURNAL SERIALIZATION
261	13	ERJ	EPFS: WAIT FOR OS REQUEST TO APPLY JOURNAL CHANGES	JOURNAL SERIALIZATION
300	22	JGG	JAVA: GARBAGE COLLECTOR GATE GUARD WAIT	JAVA (OTHER)
301	22	JAB	JAVA: GARBAGE COLLECTOR ABORT WAIT	JAVA (OTHER)
302	21	JWH	JAVA: GARBAGE COLLECTOR WAIT HANDSHAKE WAIT	JAVA JVM
303	21	JPH	JAVA: PRIMARY GC THREAD WAIT FOR HELPER THREADS DURING SWEEP	JAVA JVM
304	20	JSG	JAVA: SYNCHRONOUS GARBAGE COLLECTOR WAIT	JAVA USER (INCLUDING LOCKS)
305	20	JSF	JAVA: SYNCHRONOUS FINALIZATION WAIT	JAVA USER (INCLUDING LOCKS)

306	21	JGW	JAVA: GARBAGE COLLECTOR WAITING FOR WORK	JAVA JVM
307	21	JFW	JAVA: FINALIZATION WAITING FOR WORK	JAVA JVM
308	21	JVW	JAVA: VERBOSE WAITING FOR WORK	JAVA JVM
309	22	JGD	JAVA: GARBAGE COLLECTION DISABLE WAIT	JAVA (OTHER)
310	22	JGE	JAVA: GARBAGE COLLECTION ENABLE WAIT	JAVA (OTHER)
320	15	SOo	COMMON MI OBJECT CHECKER: SEIZE OBJECT	SEIZE CONTENTION
321	15	SOi	COMMON MI OBJECT CHECKER: SEIZE FOR IPL NUMBER CHECK	SEIZE CONTENTION
330	31	EMw	MI EVENT WAIT	MI WAIT ON EVENTS
340	8	QMr	IDLE WAIT, MI RESPONSE QUEUE WAIT	Idle / waiting for work
341	29	QMd	DATA QUEUE WAIT	DATA QUEUE RECEIVES
342	30	QMo	OTHER MI QUEUE WAIT	MI QUEUE (OTHER)

Getting started writing queries over Job Watcher collection data

Only data for the job threads and tasks *that you include in a Job Watcher collection* will be captured in the QAPYJW* files. If you do not collect data for all jobs, threads, and tasks on the system, then no system-wide data is available.

The main file for Job Watcher data is the QAPYJWTDE file.

The master file QAPYJWTDE for jobs, threads, and tasks

The master file in any collection is the TDE file called QAPYJWTDE and contains data specific to the job, thread, or task. A record is written to this file when:

- ▶ Any task count (in a selected job, thread, or task) uses CPU in an interval.
- ▶ The task count is the primary (initial) thread in a selected job and it did not use CPU in the interval, but one or more secondary threads in the same job did use CPU in the interval.

If a task count does not use CPU in an interval and it is not a primary thread, then by default no record will be written to this file for that TDE in that interval. For interval number 1, the user can optionally specify that all jobs, threads, or tasks be included in this file, regardless of their CPU usage within interval 1. This can be useful for analysis of waits that were already in progress before Job Watcher was started.

There are two main types of data stored in this file:

- ▶ Delta statistics (change since last interval snapshot)
- ▶ Cumulative statistics (from the earliest collection interval through the most recent interval snapshot)

It is, therefore, necessary to understand the content of fields in this file before running queries over it. This field level explanation is provided in Table B-6.

Table B-6 QAPYJWTDE file details

Field	Description	Comments/usage
INTERVAL	Interval number	See note a on page 214 for an example from a collection with 5-second intervals on a very busy system. This value will not be precisely equal to the collection interval.
TASKCOUNT	Task count	If this TDE represents a job's primary or only thread (TDETYPE = P), then: <ul style="list-style-type: none"> ▶ If TASKCOUNT = ITASKCOUNT, then this TDE is a primary thread in a job. ▶ If TASKCOUNT is not equal to ITASKCOUNT, then this TDE is a secondary thread in a job.

Field	Description	Comments/usage
TDEUSECS	Elapsed interval time in microseconds	The delta time between consecutive snapshots. This value should be close to the interval selected at Job Watcher collection initiation, but not identical. This value should be close but not exactly the same for all intervals in the collection.
STARTOD	Time of day at snapshot start	
STARTUSECS	Microseconds since IPL at snapshot start	
ENDUSECS	Microseconds since IPL at snapshot end	
THREADID	Thread ID in hex	The thread ID for the task count in TASKCOUNT.
ITASKCOUNT	Job initial (primary) thread task count	If the TDETYPE is not T (the record is for a thread in a job) this is the task count of the primary thread in the job. Always 0 for a task (field TDETYPE = T).
TDEJOBNAME	Job or task name	If this record is for a task count that is in a job, then this is the job name concatenated with user name (at job initiation) concatenated with job number. If this record is for a task count that is a task, then this is the system licensed internal code (SLIC) task name.
THRDSTATUS	Thread status	<p>If the task count is for a thread in a job, then this is the associated primary (initial) thread. This field does <i>not</i> apply to LIC tasks. If the contents are not blank for LIC tasks then they should be ignored. For a list of all possible values and their meaning, see the WRKACTJOB command Status field.</p> <p>Some of the most common values are:</p> <ul style="list-style-type: none"> ▶ CMNW: Wait on completion of an I/O operation to a communications device ▶ CMTW: Wait for completion of a save-while-active checkpoint being processed by another job ▶ CNDW: Wait on handle-based condition ▶ CPCW: Wait on completion of a CPI communications call ▶ DEQW: Wait on completion of a dequeue operation ▶ DLYW: Wait for a DLYJOB to complete ▶ DSC: Job has been disconnected from a workstation display ▶ DSPW: Wait for input from a work station display ▶ END: Job ended (*immed or *cntrld and delay time expired) ▶ EOFW: Wait for read after end of file ▶ EOJ: Job is ending for a reason other than ENDJOB or ENDSBS (such as SIGNOFF, ENDGRPJOB, or unhandled exception) ▶ EVTW: Wait on event (for example, QLU and SCPF waiting on event indicating work) ▶ GRP: Job is suspended due to TFRGRPJOB ▶ HLD: Job is held ▶ HLDT: Initial thread is held ▶ ICFW: Wait for completion of an I/O operation to an ICF file ▶ INEL: Initial thread is ineligible (not in activity level) ▶ JVAW: Wait for completion of a Java program operation ▶ LCKW: Lock wait ▶ LSPW: Wait for lock space to be attached ▶ MSGW: Wait on message queue for a message ▶ MTXW: Wait on mutex ▶ OPTW: Wait for completion of I/O operation to an optical device ▶ PRTW: Wait for completion of printer output ▶ PSRW: Wait in prestart job for program start request

Field	Description	Comments/usage
THRDSTATUS (continued)	Thread status (continued)	<ul style="list-style-type: none"> ▶ RUN: For a job-thread, holding an activity level (may not be using CPU) ▶ SELW: In sockets select wait ▶ SEMW: Wait for semaphore ▶ SIGS: Stopped by a signal ▶ SIGW: Wait for a signal ▶ SRQ: The suspended half of a system request pair ▶ SVFW: Wait for completion of a save file operation ▶ TAPW: Wait for completion of an I/O operation to a tape device ▶ THDW: Wait for another thread to complete an operation ▶ TIMW: Wait for a time interval to end
CURRUP	Current user profile	The real user profile for which the thread (task count) is doing work. This may be different than the user profile at job initiation, especially when using an i5/OS prestarted job.
BIRTHDAY	Job-thread (OS level) or task birth time of day	
EXTENDER	Job name extender	
TDETYPE	Job or task flag	Values are: <ul style="list-style-type: none"> ▶ P = The primary (initial) thread in a job ▶ S = A secondary thread in a job ▶ T = Task
DELTACPU	CPU time in microseconds	This value can be 0 if the TDE is the primary thread in a job and at least one of the secondary threads in the same job did use some (> 0) CPU in the interval. This is the actual CPU time used by the <i>thread</i> since the last snapshot. The value may also be zero if you specified force all TDEs in the first interval option when starting the collection. ^b
ORIGPRI	Original priority	LIC priority at job/thread/task initiation. Unless needed by IBM Service, any value shown in this field can be ignored. See Figure B-1 on page 219.
PRIORITY	Current LIC priority	LIC task dispatching priority at snapshot time. See Figure B-1 on page 219.
THREADPRI	Current i5/OS ("XPF") priority	i5/OS job/thread priority at snapshot time. If i5/OS dynamic priority scheduler adjustment is active (QDYNPTYSCD system value is 1), this value may be different than the RUNPTY field value shown through the Work with Active Jobs command. For example, an overly high CPU utilization job with RUNPTY(20) may periodically show here as 23. See Figure B-1 on page 219.
PRICHG	Priority change flag	This would be set if the job's run priority was changed during the interval, for example, by using the Change Job (CHGJOB) command Run Priority (RUNPTY) parameter.
POOL	Pool ID	
POOLCHG	Pool change flag	Job/thread moved between pools
TOTWRT	Total DASD writes	See b on page 214.
SYNDBRD	Synchronous database reads	See b on page 214.

Field	Description	Comments/usage
SYNNDBRD	Synchronous non-database reads	See b on page 214
SYNDBWRT	Synchronous database writes	See b on page 214
SYNNDBWRT	Synchronous non-database writes	See b on page 214
ASYDBRD	Asynchronous database reads	See b on page 214
ASYNDBRD	Asynchronous non-database reads	See b on page 214
ASYDBWRT	Asynchronous database writes	See b on page 214
ASYNDBWRT	Asynchronous non-database writes	See b on page 214
IOPENDING	I/O pending page faults	See b on page 214
SMSYNCIO	Waits for asynchronous writes	See b on page 214
FLTS	Page faults resulting in DASD reads	See b on page 214
ALLOCATED	Allocated DASD pages	See b on page 214
DEALLOCED	Deallocated DASD pages	See b on page 214
ALLOCATEDT	Total allocated DASD pages since job, thread, or task start	See c on page 214
DEALLOCEDT	Total deallocated DASD pages since job, thread, or task start	See c on page 214
SEIZE	Seize time in microseconds	See b on page 214
BINOVER	Binary overflow count	See b on page 214

Field	Description	Comments/usage
DECOVER	Decimal overflow count	See b on page 214
FLOATOVER	Float overflow count	See b on page 214
STMFRD	Stream file reads	See b on page 214
STMFWRD	Stream file writes	See b on page 214
MUTEX	Mutex time in microseconds	See b on page 214
ACTWAIT	Active-to-wait transitions	See b on page 214
WAITINEL	Wait-to-ineligible transitions	See b on page 214
ACTINEL	Active-to-ineligible transitions	See b on page 214
QCOUNT01 to QCOUNT32	Count of all waits in queueing bucket nn, where nn is 1 to 32	QCOUNTnn is the count of all low-level waits (ENUMs) that map into queueing bucket nn. This field is paired with the QTIMEnn field for bucket nn. ^b See Table 2-3 on page 30 for a description of wait buckets 1 through 32.
QTIME01 to QTIME32	Microseconds time for waits in queueing bucket nn, where nn is 1 to 32	The time in microseconds for all of the ENUMs (low-level waits) that map into queueing bucket nn. This field is paired with the QCOUNTnn field for bucket nn. The QTIME01 field is the dispatched CPU time, which will usually be greater than the actual CPU time used in field DELTACPU. ^b See Table 2-3 on page 30 for a description of wait buckets 1 through 32.
CURRSTATE	Current state	The state of the TDE at the end of interval snapshot time, from the XPF perspective. Possible values of: <ul style="list-style-type: none"> ► CPUQ: CPU queueing ► RUN: If task count is a job, then the job's primary thread is in an activity level. It may still be in a low-level wait such as a seize but this must be identified from the BLOCKBCKT and BLOCKENUM fields in this file. ► WAIT: The task count is in a high-level wait such as communications wait, or lock wait.
BLOCKBCKT	Current or last queueing (block) bucket	The queueing bucket identifier for the last wait that occurred, irrespective of whether the wait has completed or is still current. For the bucket description, join to file QAPYJWBKT using: SELECT blockbckt, bucketdesc FROM qapyjwtdc join qapyjwbkt on qapyjwtdc.blockbckt = qapyjwbkt.bucketnum
LICWO	Current or last low-level wait type	The four-character field is used to identify a low-level wait. This appears as three characters in all Job Watcher output through V5R3 as currently implemented, and the fourth position is always blank. This value is used for identifying the various types of low-level waits.
LICWOHNDL	Current or last LIC wait object handle	The hexadecimal representation of the 8-byte virtual address of the wait object, used by IBM Service.

Field	Description	Comments/usage
WSEGTYPE	Wait object segment type in hex	Used by IBM Service.
WOBASSEG	Wait object base segment address in hex	Used by IBM Service.
WOBJTYPE	Wait object type in hex	The internal type of the object being waited on. A list of all object types and their description is in file QIDRGUI/QAIDROT.
WOBJNAM	Wait object name	The name of the object being waited on.
WOBJTYPD	Wait object type description	The description of the type of the object being waited on, such as journal or user queue.
WSEGTYPEPD	Wait object segment type description	The LIC segment type of the object being waited on. A list of segment types and their description is in file QIDRGUI/QAIDRST.
HTASKCNT	Holding job thread/task task count	The task count of the job thread or task that is causing the current or last wait for the task count in TASKCOUNT.
HTYPE	Holding job thread/task type	Values are: <ul style="list-style-type: none"> ► P = Primary thread in a job ► S = Secondary thread in a job ► T = Task
HTASKNAME	Holding job thread/task name	The job name, job user, and job number if HTYPE is P The task name if HTYPE is T
BLOCKENUM	Current or last blocking ENUM	The ENUM for the last wait that occurred, irrespective of whether the wait has completed (last) or is ongoing (current). To get the ENUM description, join to file QAIDRJWENM using: SELECT blockenum, enumdesc FROM qapyjwtdc join qidrwch/qaidrjwenm on qapyjwtdc.blockenum = qaidrjwenm.enum
CURRWTDUR	Total time in current wait in microseconds	The time the thread or task has been in the most recent low-level wait. This value represents the wait time for <i>one</i> of the following: <ul style="list-style-type: none"> ► The thread or task was not in a wait at the end of the interval. This is the time for the <i>last</i> satisfied (ended) wait in the interval. This wait ended in the interval and then the thread/task used some CPU and did not enter another wait in the same interval. ► The thread or task was in a wait at the end of the interval. This is the time for the wait at interval end. Hence it is the time for the <i>current</i> ongoing wait (that has not yet completed). See note b on page 214 for more comments on this value.
RECCNFLCT	Relative record number if database record lock conflict	If the current wait is a DB record lock due to a read or an update (ENUM of 110 or 111 respectively; eye catcher of RDr or RDu respectively), then this is the relative record number of the record being locked.
DFTSOCKD	Default socket descriptor	

Field	Description	Comments/usage
DFTSOCKTO D	Default socket time of day	
DFTSOCKH	Default socket handle	
DFTSOCKCLV	Default socket cache level	
LISOCKD	Listen socket descriptor	
LISOCKTOD	Listen socket time of day	
LISOCKH	Listen socket handle	
LISOCKCLV	Listen socket cache level	
FRMESTOL	New mainstore frames stolen	
SREMOVE	Successful removes	
PSAINUSE	PSA entries in use	Used for SQL analysis, if SQL data was collected.
SQLINTHRD	SQL statement in progress	Used for SQL analysis, if SQL data was collected. A non-zero value indicates that an SQL statement was in progress as the interval end boundary.

- a. Intervals are hardly ever precisely the time value you specified. They are identical on a lightly loaded system. However, an interval will not be exactly 5,000,000 microseconds apart for a collection specifying 5-second intervals on a very busy system. For example, the following data captured on a customer system shows that some intervals can be considerably larger than 5 seconds (see interval 1124). Note also that intervals 1125 through 1127 are missing; this is because the task count did not use any CPU in those intervals and the task count is not a primary thread in a job where any secondary thread did use CPU.

In order to see what happened in the missing intervals, it is necessary to investigate the QAPYJWSTS file. We show a query results example that helps identify missing interval data table note a on page 215, for file QAPYJWSTS (Table B-7 on page 215).

Elapsed Interval number	Task count	interval time in microsecs	Time of day at snapshot start	Current or last state	Current or last blocking bucket
1,122	163	5,096,447	2004-08-18-22.09.32.286000	WAIT	18
1,123	163	4,933,635	2004-08-18-22.09.37.220000	WAIT	18
1,124	163	8,078,899	2004-08-18-22.09.45.299000	WAIT	18
1,128	163	5,072,306	2004-08-18-22.10.02.474000	WAIT	18
1,129	163	4,983,695	2004-08-18-22.10.07.458000	WAIT	18

- b. Delta value since last entry in this file for the same TDE.
- c. A record is produced in the QAPYJWTDE file only if the TDE used > 0 CPU in an interval or if it is the primary thread in a job. This value may be greater than the interval size if a wait spanned multiple intervals before it was satisfied (ended). However, it will never be greater than two interval sizes. If there are gaps (missing intervals) for a TDE in the QAPYJWTDE file (for example, there are records for interval 12 and 15 for a TDE), then CURRWTDUR will be the sum of the wait at the end of interval 12 and at the start of interval 15 but will not include the wait for intervals 13 and 14. If a TDE went into a wait that did not end before the collection ended and that wait occurred before the last interval in the collection, then the CURRWTDUR does not include the time spent waiting in each of the intervals prior to the collection ending. This information is captured in the QAPYJWSTS file (which has one record for every TDE for every interval) but, as of this book's publication, is not included in the reports and graphs.

Status file QAPYJWSTS

The status file QAPYJWSTS always contains one record for each TDE in the collection for every interval. The fields in this file are explained in Table B-7.

Table B-7 QAPYJWSTS file detail

Field	Description	Comments and usage
INTERVAL	Interval number	
TASKCOUNT	Task count	If this TDE is a job, (TDETYPE = P) then: ► If TASKCOUNT = ITASKCOUNT, then this TDE is a primary thread in a job. ► If TASKCOUNT <> ITASKCOUNT, the this TDE is a secondary thread in a job.
TDESTATUS	TDE status	The status of the TDE <i>during</i> the interval. This may be: ► A - Active: The task count used CPU at some time during the interval. ► I - Idle: The task count did not use any CPU during the interval. ► T - Terminated: The task count terminated during the interval.
CURWAIT	Current wait	This field contains the wait value that is the most recently entered wait state. This current wait value is not updated in a subsequent interval if the wait has completed and no new wait state has been entered. The value is changed only if a new wait state occurs. To get the ENUM description, join to file QAIDRJWENM using: SELECT blockenum, enumdesc FROM qapyjwtdc join qidrwch/qaidrjwenm on qapyjwtdc.blockenum = qaidrjwenm.enum
CURWAITD	Current wait duration in microseconds	If this value is 0, then there was no wait in progress at snapshot time (end of interval); the value in CURWAIT corresponds to the ENUM for the last (satisfied) wait. If this value is > 0, then this is the total time from the start of the current wait to the end of interval snapshot. The start of the wait may have occurred several intervals beforehand. See note a for further details.

- a. Although your collection may have specified an interval size of 5 seconds, no actual interval in the collection will be precisely 5.00000 seconds long. During an interval where active work may be relatively low to moderate, the interval could be almost exactly 5.00000 seconds. However, some may be considerably longer on a very busy system, due to the amount of work that must be done by the collector. Consider the following example for QAPYJWSTS for a task count that was idle for three consecutive intervals:

Interval number	Task count	TDE status	Current wait	Current wait duration
1,124	163	A	6	5,429,467
1,125	163	I	6	7,433,271
1,126	163	I	6	12,450,781
1,127	163	I	6	17,531,740
1,128	163	A	6	68,042

This data from the QAPYJWSTS file clearly shows that the current wait increased by more than 5,000,000 microseconds between intervals 1125 and 1126; this is because the collector snapshots are not precisely 5 seconds apart. In our collection example, using a 5-second interval, the wait time value shown as more than 5 seconds is based on how busy the system was during the interval. As illustrated in “The Job Watcher collection process in detail” on page 257, the time Job Watcher collection takes to record interval information can be lengthened significantly on a system with its processors close to 100% utilized.

In footnote a on page 215 (shown above), the 17.5-second cumulative wait at interval 1127 is not captured in the QAPYJWTDE file until the wait ends, the task count uses >0 CPU, and the delta counts and delta times are calculated for the wait buckets and written to the interval/task count record in the QAPYJWTDE file.

Job file QAPYJWPRC

The process (job) file QAPYJWPRC always contains one record for each *job* in the collection for every interval. The data is summarized for all threads in a job and stored in this file by interval and primary task count in the job. Data for tasks is not stored in this file.

The fields in this file are shown in Table B-8.

Table B-8 QAPYJWPRC file detail

Field	Description	Comments and usage
INTERVAL	Interval number	
TASKCOUNT	Initial (primary) thread task count	
JOBSBS	Job subsystem	
JOBTYP	Job type	The same job type as shown using the WRKACTJOB command. Review the WRKACTJOB command help text for more details.
JOBFNCTN	Job function	The same job function as shown using the WRKACTJOB command. Review the WRKACTJOB command help text for more details.
JOBSTATUS	Job status	Taken from the work control block and not from any threads in the job. The status of the primary thread in the job. The same job status as shown using the WRKACTJOB command. Review the WRKACTJOB command help text for more details.
DELTAPRCPU	Job CPU in microseconds	This field is the accumulation of the delta CPU values for a primary thread and all of its related secondary threads, if any, for the interval. This contrasts to the DELTACPU field in the QAPYJWTDE file, which represents the delta time value for a single thread within the interval. Because Job Watcher is a sampling tool, there could a slight difference between the value for DELTAPRCPU and summing up the individual thread DELTACPU fields in QAPYJWTDE for the same interval.
ACTTHREADD	Delta thread count	Shows the change in the number of threads existing for this job since the previous interval, if any. The thread need not be actually active during the interval, but it has to at least exist for the computation of this field and the ACTTHREADC field.
ACTTHREADC	Total threads count for the job	The total number of threads existing for this job at the interval sample time. The thread need not be actually active during the interval. Rather the thread has to at least exist for the computation of this field and the ACTTHREADD field.
CRTHREADD	Threads created during the interval	Shows the number of threads created during the interval. The threads may or may not exist at the end of the interval.
CRTHREADC	Total threads created since job start	Shows the number of threads created since the job started. The threads may or may not exist at the end of the interval.
LDIOWRT	Logical Disk I/O (LDIO) writes	The count of calls to i5/OS database programs that handle writes, such as QDBPUT, QDBPUTDR, QDBPUTM, and QDBPUTMX. See a on page 218.
LDIORD	LDIO reads	The count of calls to i5/OS database programs that handle reads, such as QDBGET, QDBGETSQ, QDBGETM, QDBGETDR, QDBGETKY, QDBGETMQO, QDBGETSQO. See a on page 218.
LDIOOTHR	LDIO other (not read/write) count	See a on page 218.

Field	Description	Comments and usage
LDIOUPD	LDIO updates	See a on page 218.
LDIODEL	LDIO deletes	See a on page 218.
LDIOFEOD	LDIO Force End of Data (FEOD) count	See a on page 218.
LDIOCOMIT	LDIO commit count	The delta number of commitment control commits since the last entry for this task count in this file. See a on page 218.
LDIOROLLB	LDIO rollback count	The number of commitment control rollbacks since the last entry for this task count in this file. See a on page 218.
LDIOOPEN	LDIO open count	See a on page 218.
LDIOCLOSE	LDIO close count	See a on page 218.
LDIOIXBLD	LDIO index build count	See a on page 218.
LDIOSORT	LDIO sort count	See a on page 218.
CMNWRT	Communication file writes	See a on page 218.
CMNRD	Communication file reads	See a on page 218.
LDTAQSND	Data queue sends	See b on page 218.
LDTAQRCV	Data queue receives	See b on page 218.
LDTAAOP	Data area operations count	
LUSRSPCIOP	User space operations or index operations count	
TXAPPIQT	Application input queueing time in microseconds	
TXINQTRAN	Application input queueing transactions	
TXRSCUT	Resource usage time microseconds	
TXRSCUTRAN	Resource usage transaction count	
TXDSPLRT	Display I/O response time microseconds	
TXDSPLTRAN	Display I/O transactions	
IFSSYMLRD	IFS symbolic link reads	
IFSDIRRD	IFS directory read count	
IFSLUCHIT	IFS lookup cache hits	

Field	Description	Comments and usage
IFSLUCMIS	IFS lookup cache misses	
IFSOPENS	IFS open count	
IFSDIRCRT	IFS directory creates	
IFSNDIRCRT	IFS non-directory creates	
IFSDIRDLT	IFS directory deletes	
IFSNDIRDLT	IFS non-directory deletes	
SOCKRD	Socket reads	
SOCKWRT	Socket writes	
SOCKBRD	Socket bytes read	
SOCKBWRT	Socket bytes written	
OPENCURSOR	Fully opened SQL cursors	
PSUCLOCURS	Pseudo-closed SQL cursors	
CURNUMACTG	Current activation group count	The current number of activation groups in the job.
CURNUMACT	Current program activation count	The current number of program activations within the job. If the same program is activated in <i>n</i> different activation groups, then this count increases by <i>n</i> .
<p>a. At the time this redbook was published, V5R3 Job Watcher QAPYJWPRC logical disk I/O fields LDIORD, LDIOWRT, LDIOOTHR, LDIOCOMIT, and LDIOROLLB contained values. LDIOOTHR includes updates, deletes, and additional values, such as opens and closes, commits, and rollbacks. The other logical disk I/O fields contain zero values until a later release when they become activated.</p> <p>b. At the time this redbook was published, V5R3 Job Watcher QAPYJWPRC fields LDTAQSND and LDTAQRCV contained zero values until a later release when they become activated.</p>		

Task dispatchable unit priorities scheme

Figure B-1 on page 219 depicts the OS-level thread priority levels (00-99) and the LIC microcode priority levels as they are assigned actual detailed priority values at the task dispatching unit level of the system. At the bottom of the figure we show excerpts of the file QPAYJWTDE query results of one of our Job Watcher collections. In most uses of the iDoctor for iSeries tools you do not need to know these low-level task priority schemes. However, these values do appear in some of the tools, so we present them here as task dispatchable unit priorities. They are considered *absolute* priority values, and are unique at any one time for every active thread and task.

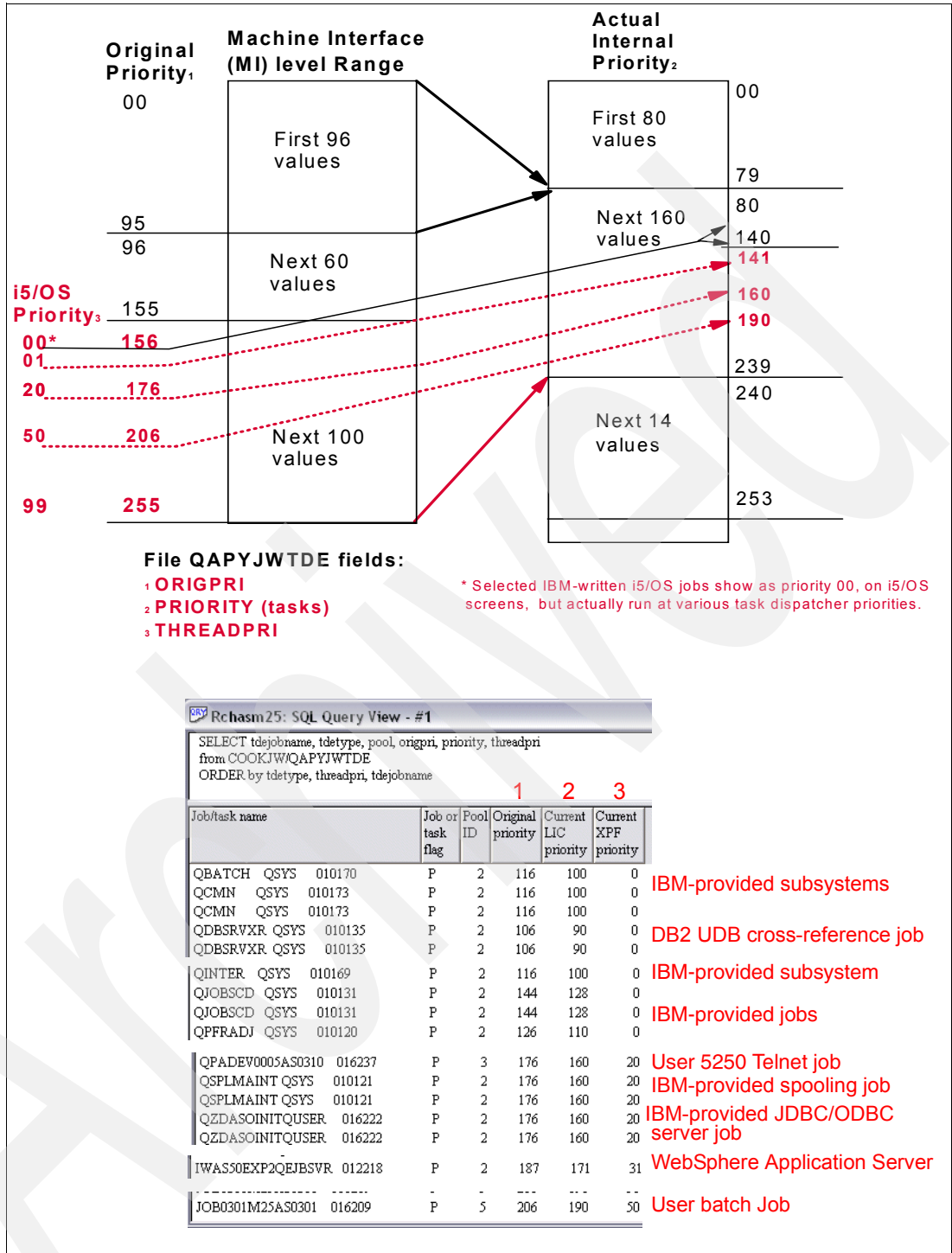


Figure B-1 Actual thread and task priority ranges with query results example

Archived



Querying and graphing tips for Job Watcher

This appendix gives you tips for running queries and graphing your Job Watcher data. It includes how to use the two Job Watcher “build SQL statements” interfaces, which enable you to query Job Watcher collection database files whether you are an SQL novice or expert.

Querying Job Watcher data

Two different SQL environments provided with Job Watcher enable you to run SQL-based queries over your existing data. They are:

- ▶ Query definition interface (for novice SQL users)
- ▶ SQL Query view (for expert SQL users)

Query definition interface

Tables and graphs within the Job Watcher component are created via an underlying query definition. The query definition defines exactly how the data is to be retrieved and from what files, then displays the results in a table or graph. The Query Definition is an interface over these existing SQL statements that enables a novice SQL user to customize the query for the active table or graph within a Data Viewer without having to be an expert at SQL. Nearly all table and graph views have a query definition menu available that lets you work with the query definition for that particular table or graph view.

It does this by providing an interface that helps you build an SQL statement to meet your criteria based on your selection of Field, Member, or Record. After you have selected your criteria, you can examine this SQL statement later to see how it was built. The Query Definition interface restricts you to working with only one file.

Note: The Data Viewer is a window within the iDoctor for iSeries suite of tools that is used for displaying tables and graphs over data on your iSeries system. You can have as many Data Viewer windows open at one time as you want. The data behind the views within a Data Viewer may come from any number of systems.

Accessing the query definition interface

There are different ways to access the Query Definition interface: You can right-click in any Data Viewer and use the query definition context menu to start the Query Definition interface, or you can select the right-most query icon displayed on the Data Viewer toolbar shown in Figure C-1 and indicated by the arrow.

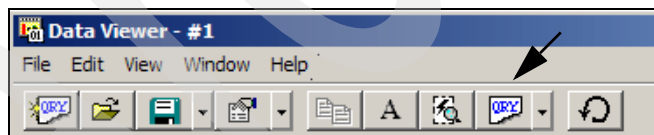


Figure C-1 Data viewer toolbar: Query Definition interface icon

If you right-click in the Data Viewer:

1. In the pop-up context menu displayed in Figure C-2, select **Member Selection** for this example.

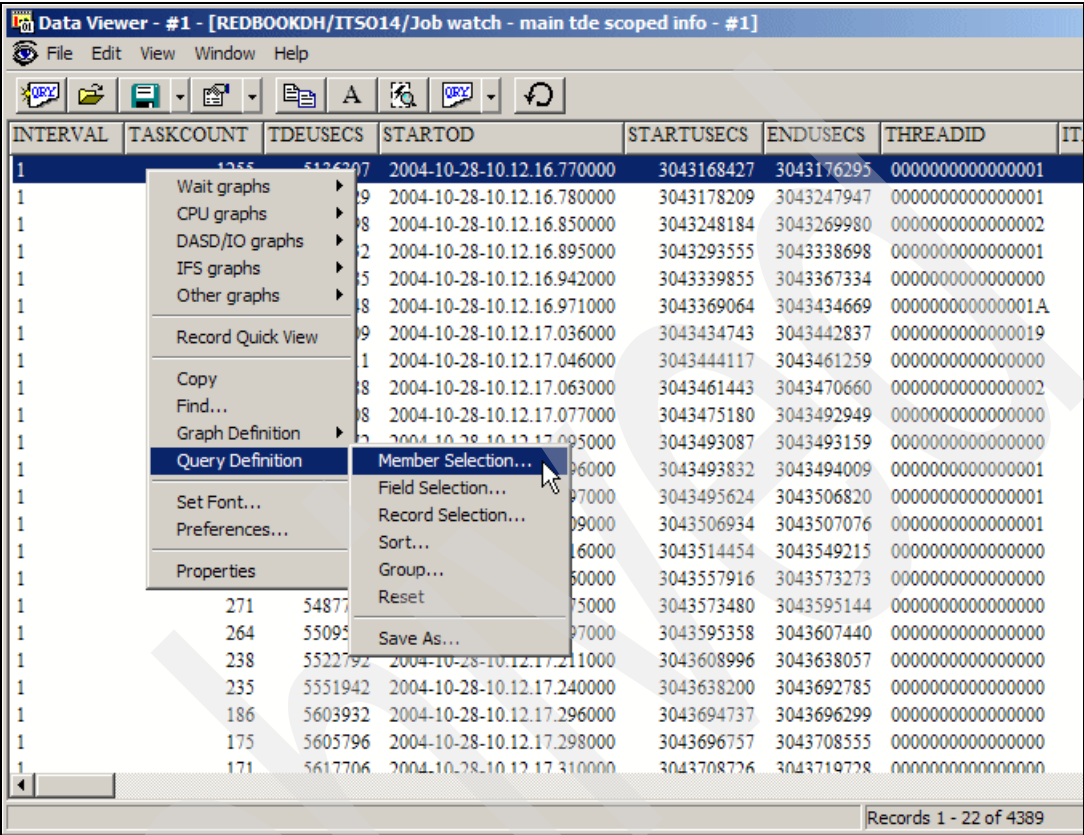


Figure C-2 Query Definition → Member Selection

This displays the Query Definition interface, which contains several different tabs for defining specific parts of the query. In our example, we chose Member Selection, opening the tab shown in Figure C-3. Member Selection enables you to view or modify the file or member of the query.

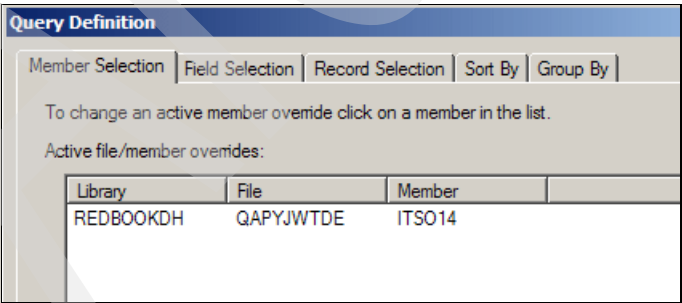


Figure C-3 Query Definition: Member Selection

2. To change an active member override, click on a member in the list to highlight it. (In this example, we only have one member.) Click the **Field Selection** tab (Figure C-4). The list of fields indicates the order and content of the data in the view. In this display, you can define which fields will be visible or hidden in the report and in what order.

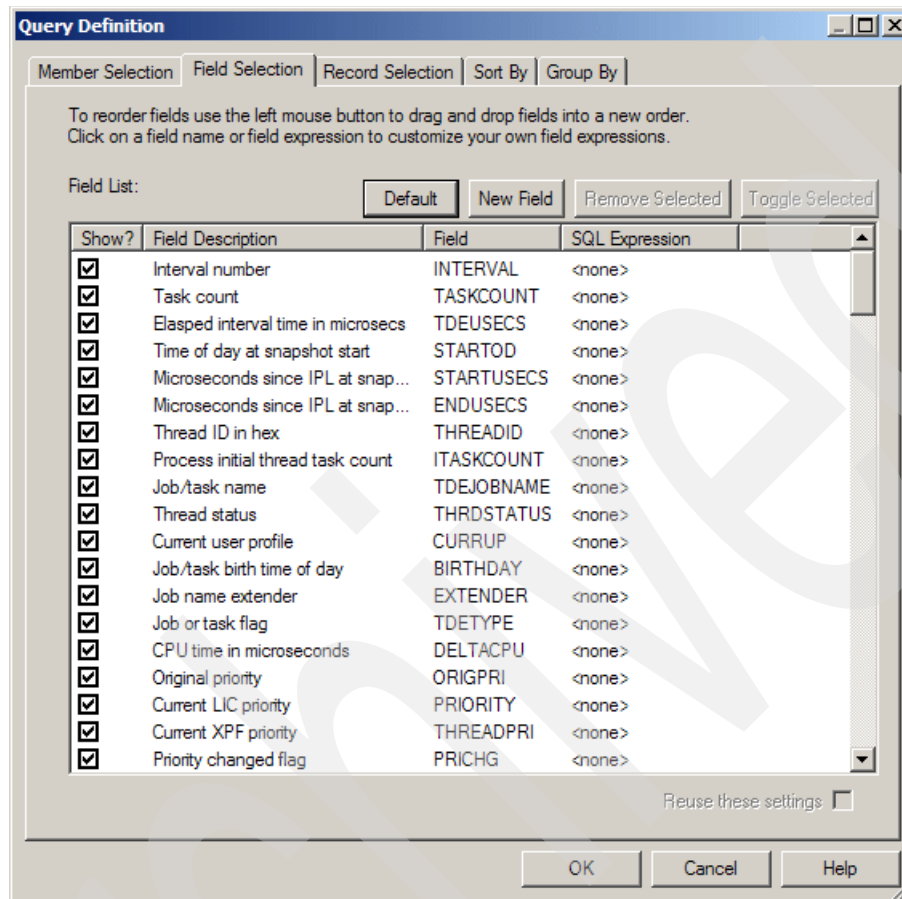


Figure C-4 Query Definition: Field Selection tab

- The order of the fields displayed in the Field List directly affects the order that the fields are displayed in the table view. To change the order of the field names, highlight one or more adjacent or non-adjacent field names and click and drag them to the desired location within the list.

For example, you might want to see CPU time in microseconds right after the task count, so you would move this field name right after task count as shown in Figure C-5.

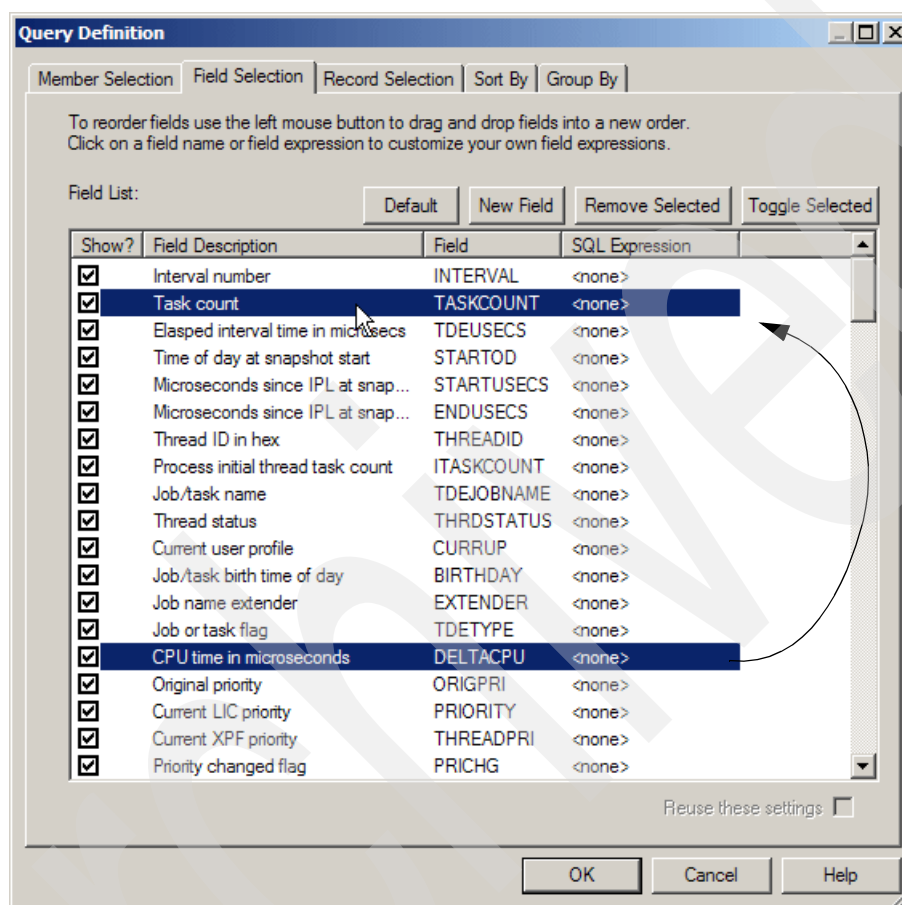


Figure C-5 Moving fields within Field selection

4. Visible fields are indicated by a check mark in the Show? column within the Field List. If a field is not checked, then it will not be shown. If you want to remove a large number of the field names, you can highlight a group of adjacent fields by clicking the first one you want to remove, holding down the Shift key, dragging your mouse to the last field name in your group, then clicking **Toggle Selected**. This toggles the check box settings (between checked and non-checked) within the group of field names that you have selected, as shown in Figure C-6. This can be very handy when you want to hide or show a large number of fields at once.

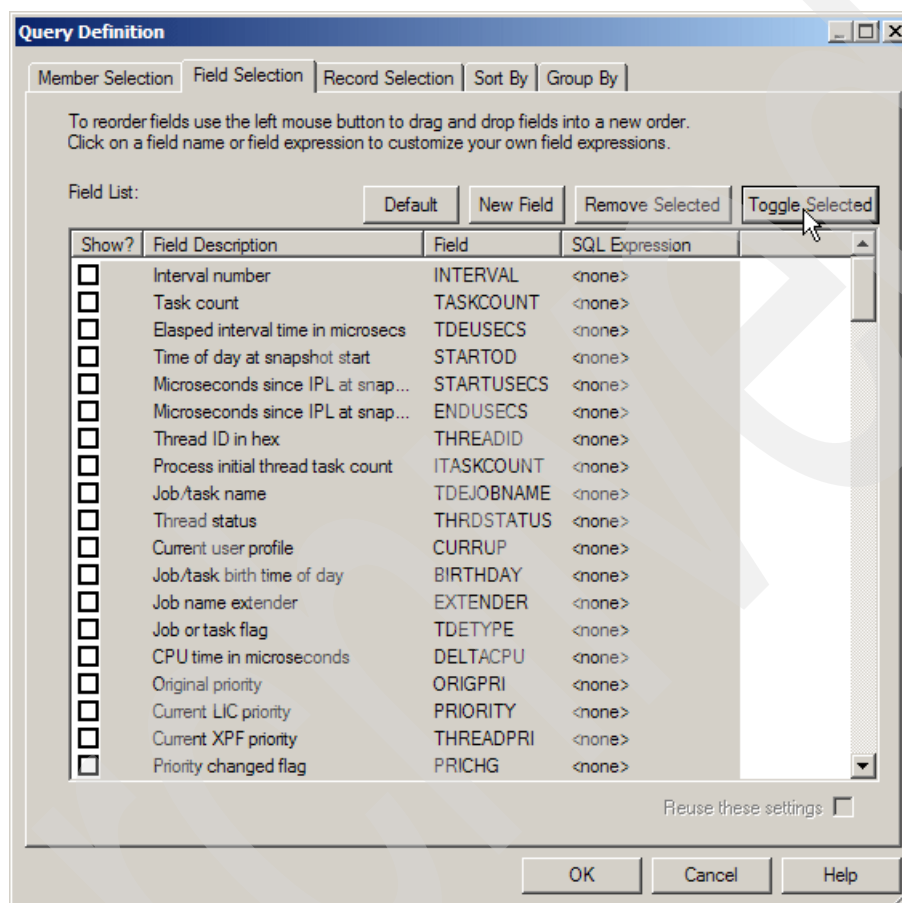


Figure C-6 Toggle selected fields

You may also use this panel to create your own fields (Figure C-7 shows the result):

- Click **New Field** to add a new row to the list of fields.
- In the description cell for the new row in the list, type a description for the new field.
- Click the field cell in the list and provide a value. This value is the short SQL field name in your SQL field list.
- Click the SQL expression cell in the list and enter your SQL expression for this field. For our example, we create a new field called JOBNAME with a description of Job Name. We want this to be a substring of the TDEJOBNAME field. We know that we want to look at the first six characters of the TDEJOBNAME because we want to investigate our nightly batch jobs, which begin with UPDADD.

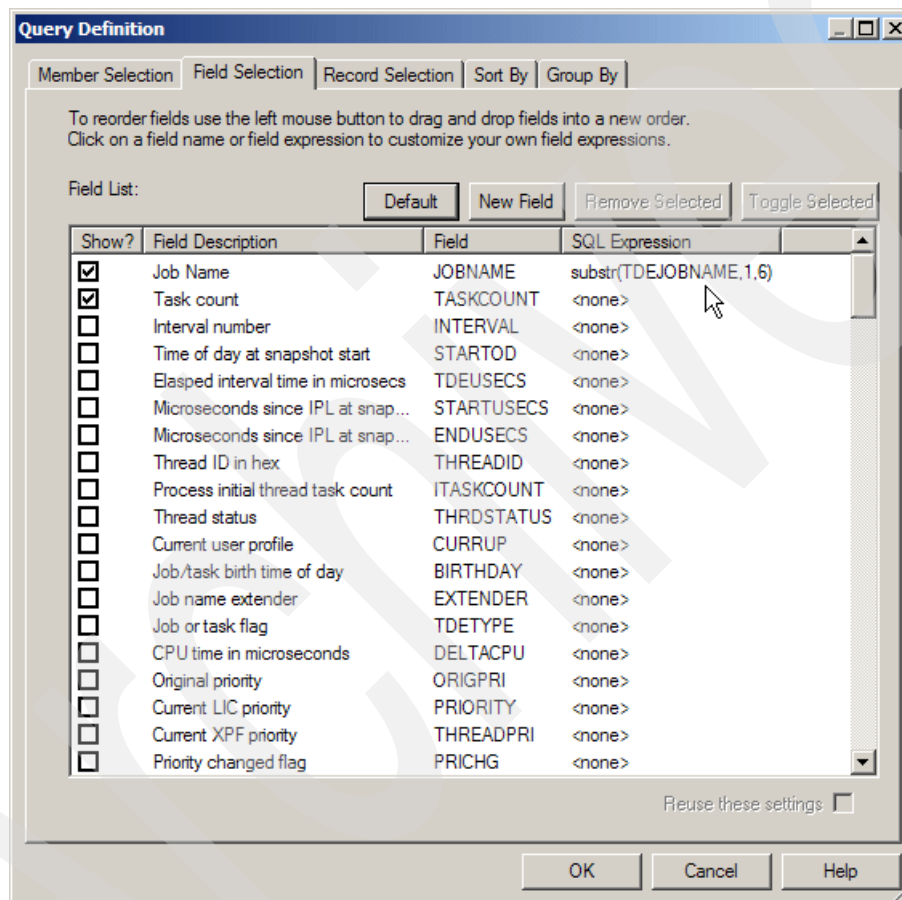


Figure C-7 Creating your own fields in Field Selection

Note: The **Reuse these settings** option at the bottom of Figure C-7 is enabled only for PEX Analyzer report files. This is for reordering the fields and this order into the current PC (in the Windows Registry). The next time the analysis report is opened, the same field ordering is used.

5. Click the **Record Selection** tab to define which records will be visible in your report.
For our example, we only want to look at the job names that equal UPDADD:
 - a. Set the Field name to the new field we created called **JOBNAME**.
 - b. From the Operator pull-down menu, set the operator to **=**
 - c. Set the Value to UPDADD.
 - d. Click **Add Filter** to add the criteria to the Record Selection Filter List (Figure C-8).

Query Definition

Member Selection | Field Selection | **Record Selection** | Sort By | Group By

Field: Add Filter

Operator: equal

Value:

Example: depends on expression type
Boolean condition:
☒ AND ☐ OR

Record Selection Filter List:

Field	Operator	Value	And/Or
Job name (JOBNAME)	=	'UPDADD'	AND

Figure C-8 Record Selection Filter List

6. Click the **Sort By** tab to define a sort sequence for your report. For our example, we want to sort the report by task count in ascending order:
 - a. Set the Field parameter to **Task count (TASKCOUNT)**.
 - b. Set Sort Order to **Ascending**.
 - c. Click **Add Field** to add the criteria to the Field Sort Sequence List as shown in Figure C-9.

Query Definition

Member Selection | Field Selection | Record Selection | **Sort By** | Group By

Field: Add Field

Sort Order:
☒ Ascending ☐ Descending

Field Sort Sequence List:

Field	Direction
Task count (TASKCOUNT)	Ascending

Figure C-9 Query Definition: Sort By tab

- Click the **Group By** tab to define which records will be visible in the report (Figure C-10). Group-by queries are valid only when the fields on the field selection page comply with the SQL rules for running Group By within an SQL statement. Any fields that are not part of the group-by clause must be summarized in order to exist in the field selection, or the query will not run. For our example, we want to group the report by JOBNAME (the new field we created) and TASKCOUNT. Choose each from the pull-down menu and click **Add Field** to add to the Group By Field list. Click **OK**.

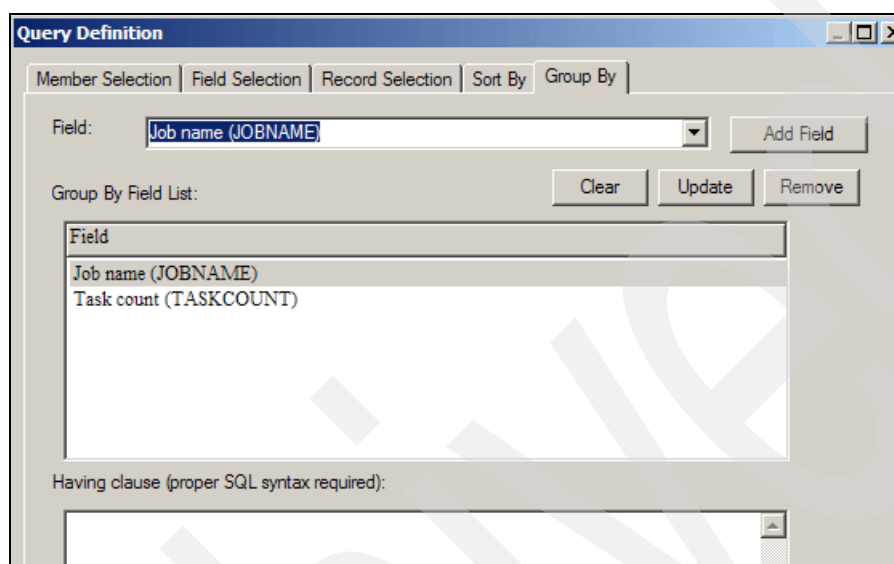


Figure C-10 Query Definition: Group By

Restriction: The Query Definition interface is built by parsing the contents of an existing SQL statement. This parsing works well for many queries but it does not acknowledge all types of SQL syntax. It will parse most SQL select statements containing JOINS but there are some very complex statements that cannot be parsed (such as UNIONS). Although a query can be parsed that contains JOINS, the type of JOINS and the files being JOINed are not changeable through the interface.

To do this through the iDoctor for iSeries GUI, use the SQL Query view described in “Accessing the SQL Query view” on page 233 and type in the SQL statement as needed. The query definition can be used to adjust the WHERE, ORDER BY, and GROUP BY clauses of the outermost part of the SQL statement. Any ORDER BY, WHERE, or GROUP BY clauses for subqueries in the SQL statement are not configurable through this interface.

The results of your user-defined query are shown in Figure C-11. We now know the task counts for each of the UPDADD nightly batch jobs that were running during our collection.

JOBNAME	TASKCOUNT
UPDADD	1307
UPDADD	1308
UPDADD	1309

Figure C-11 Query definition results

Viewing your query

You can look at your query definition by right-clicking in an active Data Viewer that displays the results of the query and selecting **Properties**, as shown in Figure C-12.

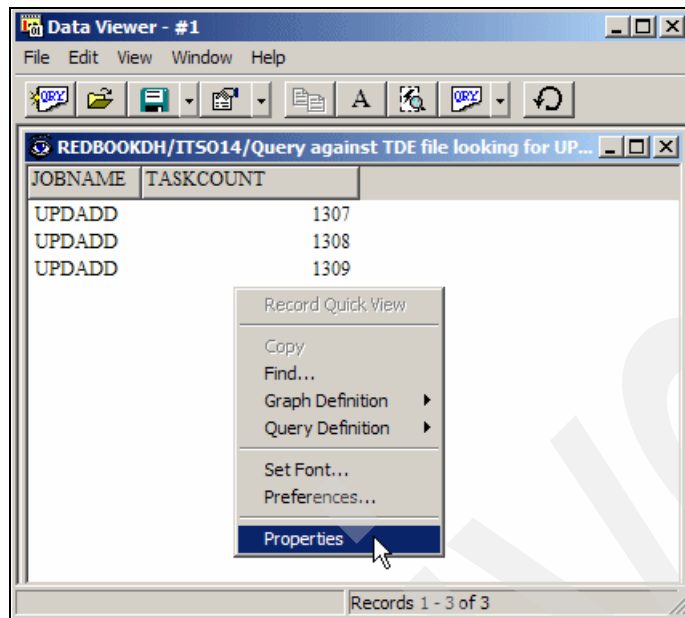


Figure C-12 Selecting Properties in the Data Viewer

Figure C-13 shows the properties of your user-defined query. If this is a query that you might want to use again with some slight modifications, you can copy the SQL statement (Ctrl + C) and paste it (Ctrl + V) into the SQL Query view and modify it to fit your needs. The SQL Query view is discussed in more detail in “SQL Query view” on page 233.

Properties

Query

Description:
REDBOOKDH/ITS014/Query against TDE file looking for UPDADD jobs

SQL Statement:
SELECT substr(TDEJOBNAME,1,6) AS JOBNAME, TASKCOUNT FROM
REDBOOKDH/QAPYJWDE WHERE substr(TDEJOBNAME,1,6) = 'UPDADD' GROUP BY
substr(TDEJOBNAME,1,6), TASKCOUNT ORDER BY TASKCOUNT ASC

Database file overrides:

Library	File	Member
REDBOOKDH	QAPYJWDE	ITS014

OK Cancel Help

Figure C-13 Properties of a user-defined query

Saving your new query definition

Save your query definition to use later by right-clicking it and choosing **Query Definition** → **Save As** in an active table view (Figure C-14). All Query Definitions are saved into the QAIDRSQLO4 file.

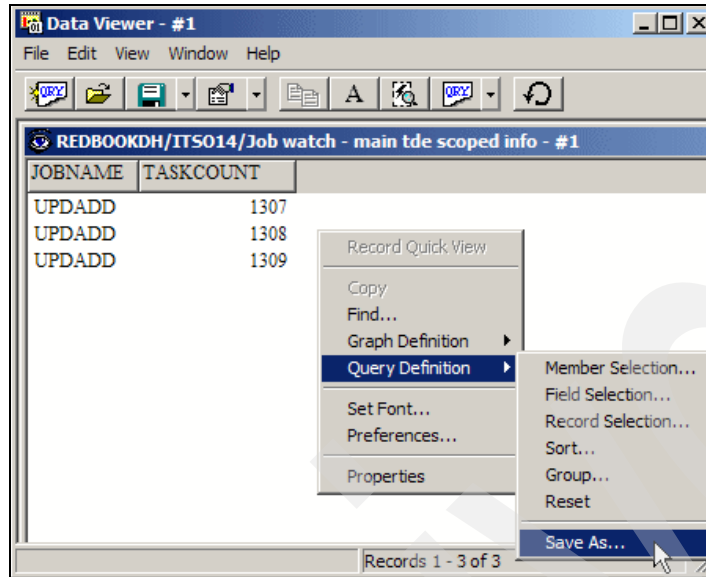


Figure C-14 Saving your new query definition

Figure C-15 shows an example of the Save Query Definition Interface. You can assign a description for your query definition up to 250 characters long.

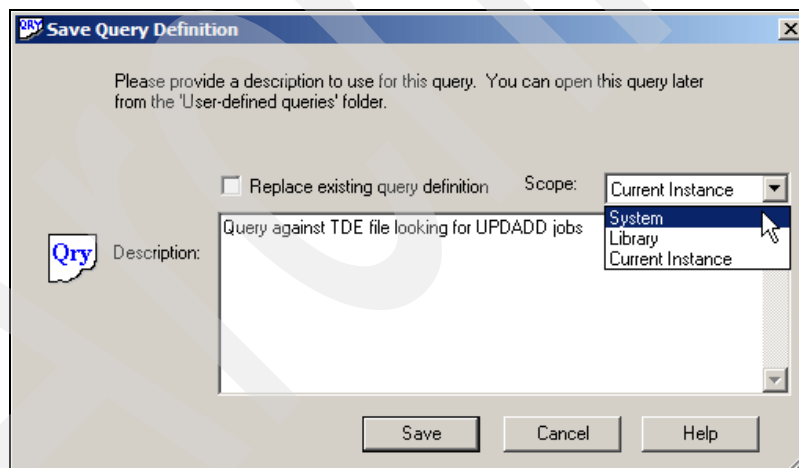


Figure C-15 Save Query Definition

The Replace existing query definition box, which replaces the saved query definition with the current one, is visible only if the table view was created from a user-defined query definition.

Use the Scope option to set the scope of the query. This determines at which level (system, library, or current instance) the query is visible. If the scope is Library or Current instance, then the query definition is contained in the QAIDRSQLO4 file in the same library as the current collection. System scoped query definitions are stored in library QUSRSYS. System scoped queries can be used against any library on the system as long as it contains the same files that the current query uses.

Working with query definitions

After a query definition has been created, it is displayed and opened from the user-defined queries folder under a collection as shown in Figure C-16.

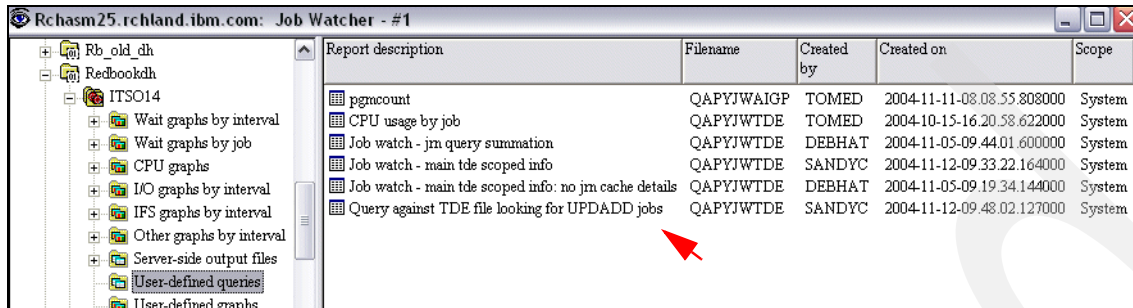


Figure C-16 Work with user-defined queries

SQL Query view

With the SQL Query view, which is for more experienced SQL users, you can dynamically execute and display the results of a query using Structured Query Language (SQL). This enables you to work with more than one file using JOINS or UNIONS; however, you must know how to write SQL as there is no assistance with building the statement in this environment.

The queries you create with this view may be saved and restored for later use and their definitions can be viewed and manipulated using the query definition interface. A benefit of using query definition with the SQL Query view is that any changes to the query definition (which changes the SQL statement) are visible immediately in the top portion of this window.

Accessing the SQL Query view

The SQL Query view can be accessed within a Data Viewer by selecting the left-most query icon from the Data Viewer toolbar, as indicated by the arrow in Figure C-17.

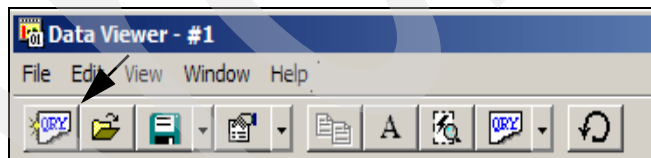


Figure C-17 Data Viewer toolbar: SQL Query view icon

When you click this button, the System selection window appears as shown in Figure C-18. Click **OK** if this is the system you want to run your SQL statements over. Otherwise, change to the appropriate system and then click **OK**.

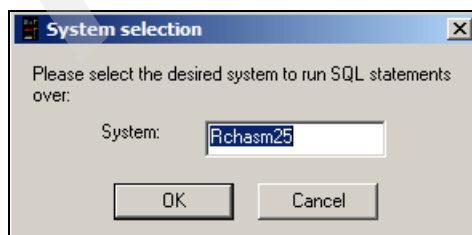


Figure C-18 System selection window

Figure C-19 shows the top portion of the view as an area where you enter SQL statements, and the bottom portion is where the results or output from your SQL statements appears after your query is executed.

To execute your query after typing it in, right-click the top portion of the view and choose **Execute** (or press F4) as shown in Figure C-19. You can re-execute your query at any time.

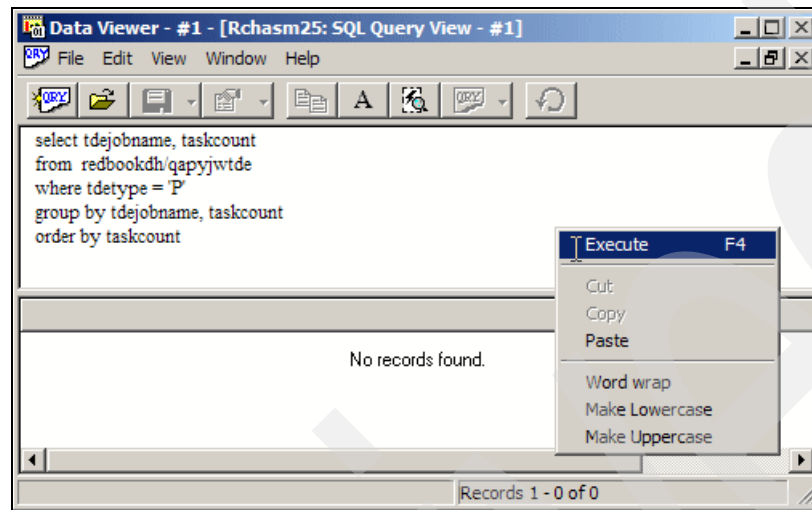


Figure C-19 Executing your SQL statements

If you have more than one Job Watch in your library, Figure C-20 appears for you to choose the desired member so the appropriate override can be performed. Highlight the desired member and click **OK**.

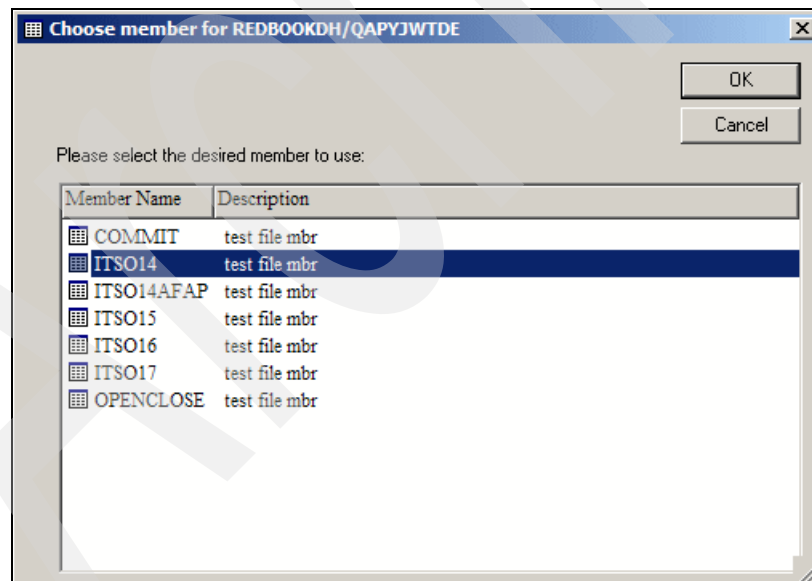
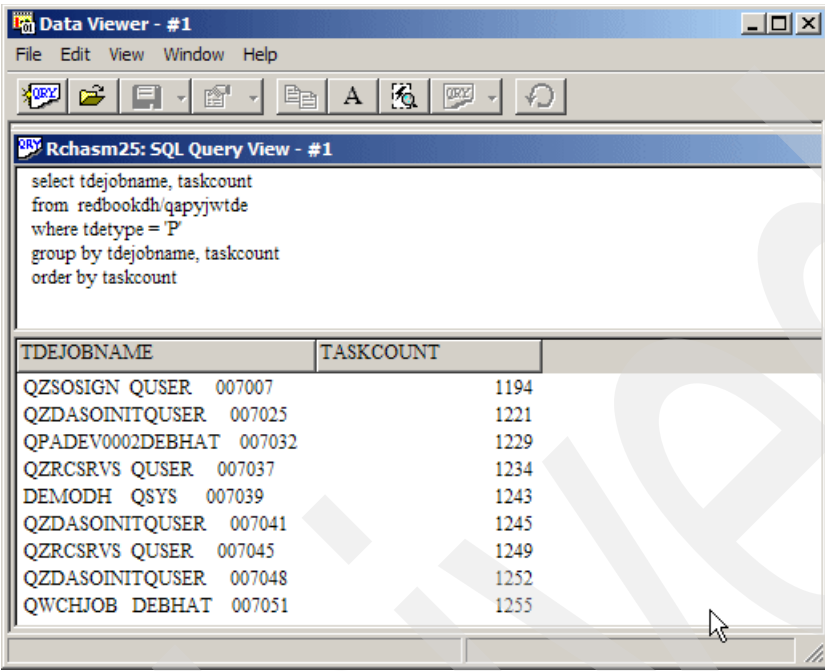


Figure C-20 Choose member for override database file

The bottom portion (Figure C-21) shows the results from your SQL statements in table view, which enables the typical copy-to-clipboard, query definition, and find functions that are standard in a table view.



Rchasm25: SQL Query View - #1

```
select tdejobname, taskcount
from redbookdh/qapyjwtdc
where tdtype = 'P'
group by tdejobname, taskcount
order by taskcount
```

TDEJOBNAME	TASKCOUNT
QZSOSIGN QUSER 007007	1194
QZDASOINITQUSER 007025	1221
QPADEV0002DEBHAT 007032	1229
QZRCRVS QUSER 007037	1234
DEM0DH QSYS 007039	1243
QZDASOINITQUSER 007041	1245
QZRCRVS QUSER 007045	1249
QZDASOINITQUSER 007048	1252
QWCHJOB DEBHAT 007051	1255

Figure C-21 Query results from SQL statements

The queries you create with this view may be saved and restored for later use, and their definitions can be viewed and manipulated using the Query Definition Interface. A benefit of using the Query Definition Interface along with the SQL Query view is that any changes to the query definition (which changes the SQL statement) will be immediately visible in the top portion of this window.

Changing field (column) headings in your query results

The query results shown in this appendix and the RPG, SQL, and Java analysis example chapters use the Field Description text associated with the Job Watcher database file field as the query results column heading text. In this appendix you have also seen how to create your own field as described for Figure C-7 on page 227.

In this topic we use the facilities shown already and go one step farther in customizing your own query results by defining our own field description text to be used as the query results column-heading text for fields derived within the SQL SELECT statement.

As an example, we use a SELECT statement seen earlier in this book (Example 5-2 on page 153) and its associated results in Figure 5-30 on page 153. We change the column heading text only for the derived field PctJrnwait, but you can follow the steps we show here for any of the fields.

Figure C-22 on page 236 shows the original SELECT statement and query results. Dashed-line boxes highlight the phrase within the statement beginning with CAST((double(sum(qtime14))) and ending with as PCTJrnwait. We want to give the derived field a more meaningful column heading in the query results.

To begin this change:

1. Right-click any row in the results area.
2. In the context menu, select **Query Definition** → **Field Selection**.

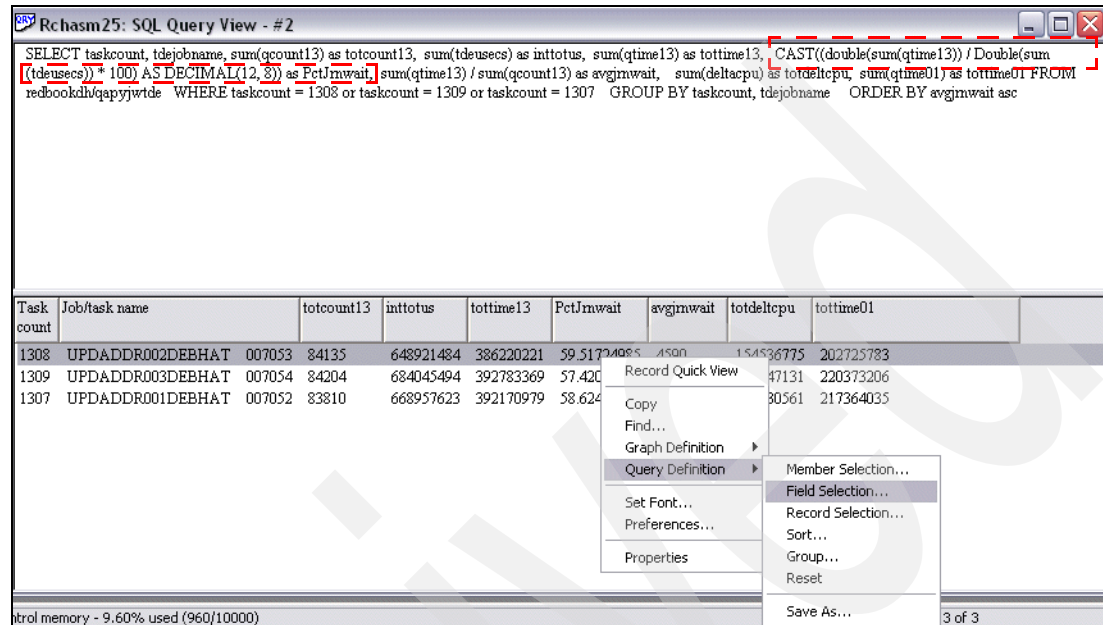


Figure C-22 SQL and query results: original column headings

3. This opens the Query Definition window, which has several functions. We show how to change the column heading text (the Field description part of the field definition).

Double-click the field you want to modify. (In our example it is **PctJrnwait**.) This opens the Query Definition: Edit Field window shown in the lower area of Figure C-23.

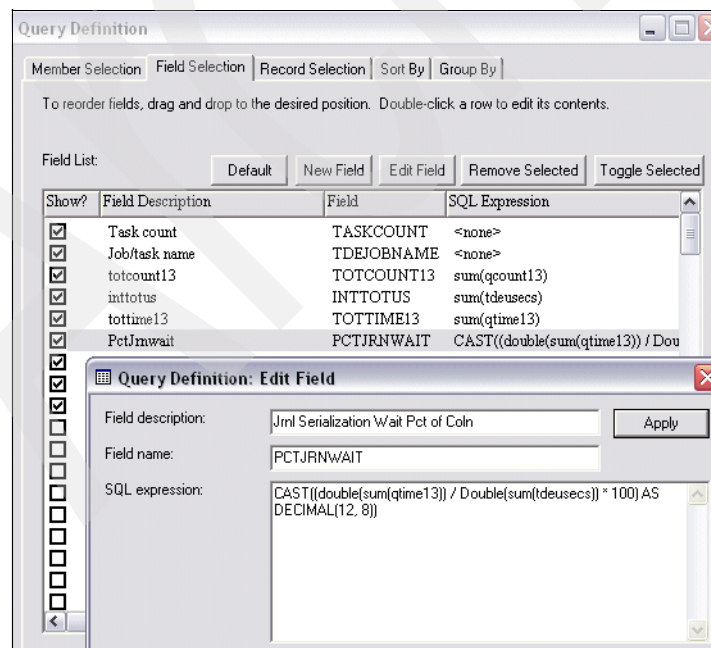


Figure C-23 Changing the derived field PctJrnwait column heading: example 1 of 3

4. Enter your new text into Field description. We used Jrnl Serialization Wait Pct of Coln (collection).
5. Press the ESC key or click the window's X button in the upper-right corner to exit and return to the Query Definition window.
6. Click **OK** on the Query Definition window to change the query and automatically rerun it.

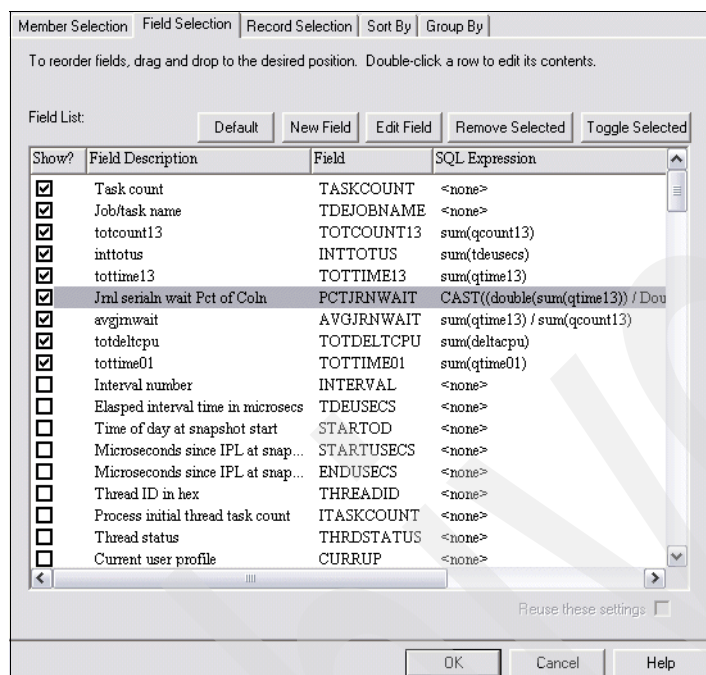


Figure C-24 Changing the derived field PctJrnwait column heading: example 2 of 3

The query results show the new column heading for derived field PctJrnwait (Figure C-25).

Rchasm25: SQL Query View - #2

SELECT TASKCOUNT, TDEJOBNAME, sum(qcount13) AS TOTCOUNT13, sum(tdeusecs) AS INTTOTUS, sum(qtime13) AS TOTTIME13, CAST(((double(sum(qtime13)) / Double(sum(tdeusecs))) * 100) AS DECIMAL(12, 8)) AS PCTJRNWAIT, sum(qtime13) / sum(qcount13) AS AVGJRNWAIT, sum(deltacpu) AS TOTDELTCPU, sum(qtime01) AS TOTTIME01 FROM REDBOOKDH/QAPYJWTD WHERE taskcount = 1308 OR taskcount = 1309 OR taskcount = 1307 GROUP BY taskcount, tdejobname ORDER BY PCTJRNWAIT ASC

Task count	Job/task name	totcount13	inttotus	tottime13	Jrnl serialn wait Pct of Coln	avgjrnwait	totdeltcpu	tottime01	
1309	UPDADDR003DEBHAT	007054	84204	684045494	392783369	57.42065000	4664	156547131	220373206
1307	UPDADDR001DEBHAT	007052	83810	668957623	392170979	58.62418866	4679	154330561	217364035
1308	UPDADDR002DEBHAT	007053	84135	648921484	386220221	59.51724985	4590	154536775	202725783

Figure C-25 Changing the derived field PctJrnwait column heading: example 3 of 3 (new heading)

You can repeat the steps shown for each derived field, if you wish.

Graph views

In Job Watcher, users can define graphs over data within any desired table view or graph view. This is done by creating a graph definition, which is the information that builds a user-defined or iDoctor-supplied graph. Like query definitions, graph definitions are stored on the server in a database file.

A graph definition defines everything needed to display the graph, including a reference to the query definition needed to produce the graph's data. Whenever a graph view is saved, the current query definition and the current graph definition are saved.

The interface is accessible by either defining a new graph definition using the **Graph Definition** → **Define New** context menu within a table view or by using the Graph Definition pop-up menu in a user-defined graph view.

User-defined graphs

As the name states, user-defined graphs are created by you, the user, and saved into a definition on the server. Its settings are called its graph definition. The SQL statement (query) behind the table view is used for the query behind the graph. Such a graph is created from a table view using the **Graph Definition** → **Define New** context menu as shown in Figure C-26.

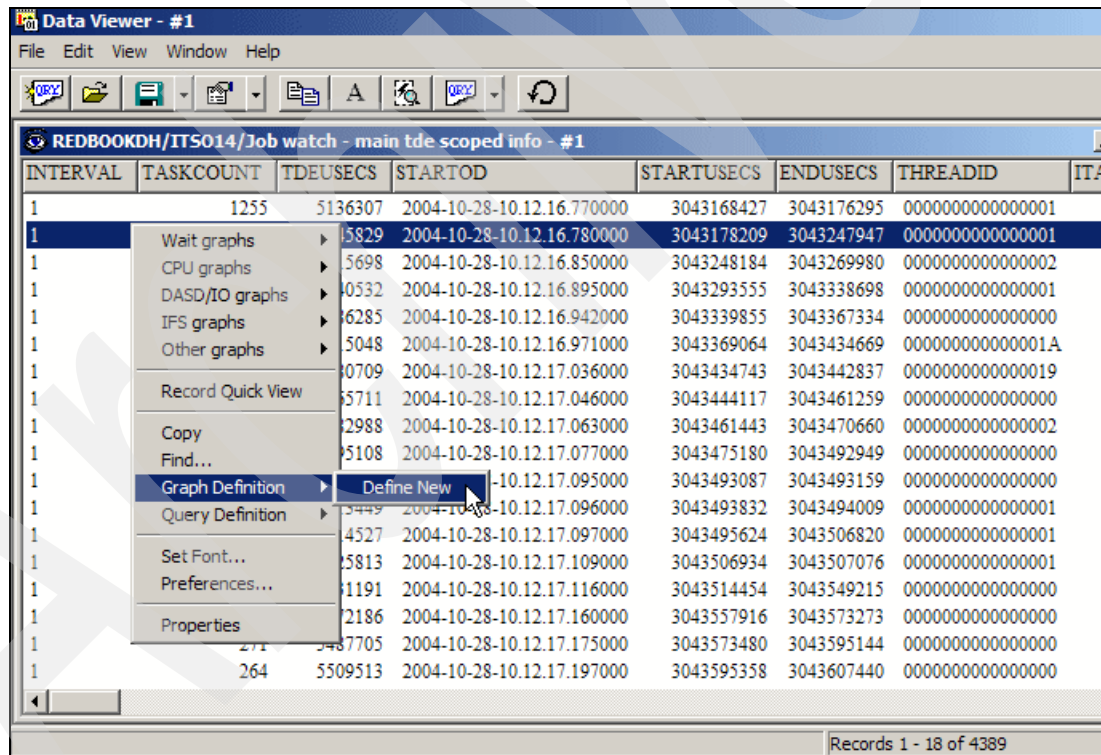


Figure C-26 Graph definition → Define New

The General/X-axis tab lets you define the graph title, type, bars per page, and the field used for the X-axis. See Figure C-27.

Figure C-27 Graph Definition: General/X-axis

Use the Primary Y-axis tab to define the fields that should be displayed on the graph. Each field represents a bar on the graph and can have a different color and customized description. See Figure C-28.

Field	Description
Seize contention time (seconds) (TIME15)	Seize contention time (seconds)
Mutex/semaphore contention time (seconds) (TIME12)	Mutex/semaphore contention ti
Object lock contention time (seconds) (TIME17)	Object lock contention time (se

Figure C-28 Graph Definition: Primary Y-axis

On the Secondary Y-axis tab (Figure C-29), you can define a bar graph's secondary Y-axis, which is always a black line over the bar graph with a Y axis on the right side of the graph. This axis is visible only for horizontal bar graphs. (Use Figure 3-76 on page 115 as an example.)

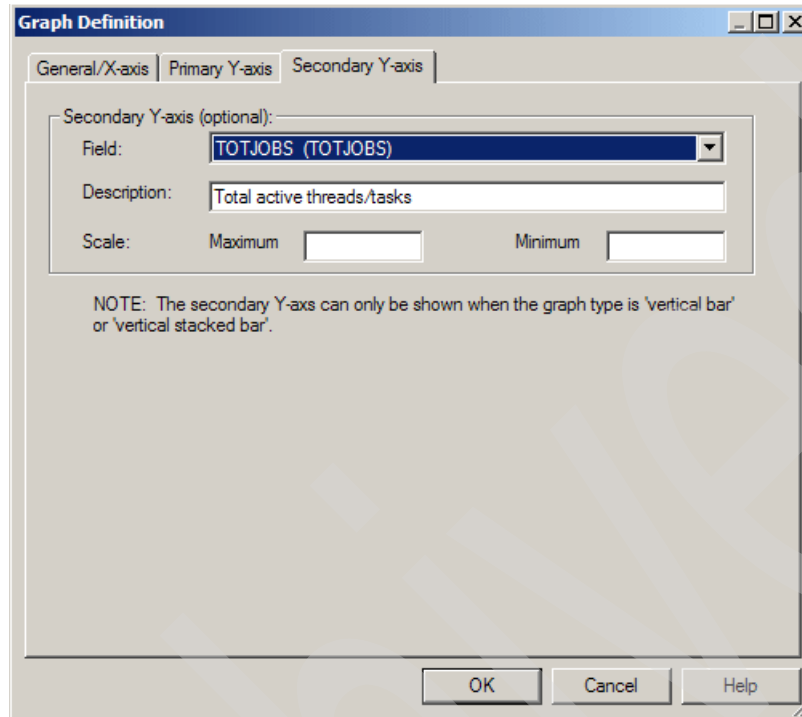


Figure C-29 Graph Definition: Secondary Y-axis

Overview of Job Watcher 5250 commands

Although the iDoctor for iSeries Job Watcher is a graphical user interface (GUI) tool used to view and collect data for jobs, tasks, and threads on a system, some 5250 (green screen) commands are provided for creating and working with Job Watcher data. This appendix gives an overview of these commands and provides source code for automatically submitting several Job Watches.

Job Watcher 5250 commands

This section contains the Job Watcher commands entered from a 5250 command line.

WCHJOB command

Use this command to collect Job Watcher data on your system. It creates the same files produced by the Job Watcher collection function initiated from the GUI. The database files are described in Appendix B, “Database files created by Job Watcher” on page 199.

Note: This command is allowed to run interactively, but we recommend that the job be submitted to batch. This is because it runs synchronously with Job Watcher and therefore can slow down collection if lots of data is collected. It is much better to submit a job or send a message to another job to do the processing.

WCHJOB parameters

OUTLIBMBR

Output files library

The name of the Job Watch collection library. The library must already exist.

Output files member

Specifies the name of the collection member. Creating a Job Watch creates several files in the output files library, each having this member name.

REPLACE - Replace members

**YES (Default)*

A member with the same name in the collection library will be replaced with the new collection member.

**NO*

A member of the same name is not replaced and the collection will fail if the specified member name already exists.

UNTIL - Ending criteria

Limit type

The possible values for the limit type are:

► *TRANSTONONE

Job Watcher collection ends when all of the watching jobs/tasks/threads end.

*TRANSTONONE is a single value and does not require the Limit value parameter.

Note: The *TRANSTONONE value is available only from a 5250 session.

► *DASDMB

Job Watcher collection ends when the total disk space consumed by the Job Watch exceeds the value specified (in megabytes).

► ***NBRSEC**

Job Watcher collection ends when the specified number of seconds has elapsed since the collection was started.

► ***NBRITV**

Job Watcher collection ends when the specified number of intervals have been collected.

Limit value

Enter a number for each.

Limit type *DASDMB = megabytes, *NBRSEC = seconds, and *NBRITV = intervals.

INTDELAY - Seconds between intervals

Time between collection intervals in seconds.

***NODELAY**

Use this option to indicate that the Job Watcher engine should begin taking another snapshot immediately after the previous snapshot completes.

JOBS - Job names (qualified/generic)

Identify the job names to include in the collection. Each entry contains a specific or generic job name. If the special value *ALLJOBS is used, then information about every job/thread on the system is collected.

Job name

This value is the specific job name, a generic job name, *ALL, or *ALLJOBS.

Job user

This value may be the specific user name, a generic user name, or *ALL.

Job number

This value is the six-digit job number or *ALL.

TASKS - Task names

Identify the system tasks to include in the collection. Tasks may be specified using exact names or generic task names. Up to 20 entries are allowed for this parameter.

CURRUSER - Current user profile name

This parameter is used to select jobs by their current user profile. Up to 20 profiles may be specified.

TASKCOUNT - Thread task count

List the task count (or counts) of the jobs/tasks to be included in the collection. This value must be entered in decimal format (not hex). Up to 20 are allowed.

CURRSBS - Current job subsystem name

Watch jobs running in the specified subsystem. Only one subsystem entry is allowed.

CURRPOOL - Current thread or task pool

Watch jobs running in the specified pool. Only one pool entry is allowed.

TSS - Thread selection string

The thread selection string is an alternate means to specify which jobs or tasks should be included in the collection.

The following parameter values must be used when defining the thread selection string: JOB, TASKNAME, CURRUSER, TASKCOUNT, CURRSBS, CURRPOOL. Each parameter value must be enclosed in single quotes for all of the parameter types listed above. The Boolean values of .AND. or .OR. should be used if multiple entries are desired in the thread selection string.

Example 6-1 Using the thread selection string

```
JOB = 'MYJOB/MYUSER/123111'.OR.CURRSBS = 'IDCTOR' CURRPOOL = '2'

TASKCOUNT = '131331112'.OR.TASKCOUNT = '13122223'

JOB = 'QZDA*/*ALL/*ALL'.OR.TASKNAME = 'RTZ*'
```

DATATYPEU - Fixed data

Data types

The following data types are available for collection:

***ACTGRPDETAIL**

Includes the activation group and program information along with heap size and blocks. Also contains *ACTGRPSUM.

***ACTGRPSUM**

Collects activation group totals. Data is found in the QAPYJWPRC file.

***SQLSTMT**

Collects last executed SQL statements and host variables.

***SQLCURSTMT**

Collects active SQL statements and host variables.

***SQLDETAIL**

Collects last executed SQL statements, host variables, prepared statement arrays, and open cursors.

***CALLSTACK**

Collects call stacks. Up to 1000 levels deep may be collected for each stack.

***SOCKETTCP**

Lists details about all sockets open to a TCP/IP connection at the job/task/thread level.

***SOCKETJOBS**

Lists all jobs/threads/tasks using the same socket. Should be used in combination with *SOCKETTCP.

Interval frequency

All fixed data types may be collected in one of the following ways:

► ***ALWAYS**

Data is collected every interval.

- ▶ ***NEVER**

Data is never collected.

Numeric value

Data is collected every Nth interval.

DATATYPEC - Dynamic data types

Data types

The following dynamic data types are available for collection.

- ▶ ***CONFLICTCALLSTACK**

Call stacks are captured when there is a holder/waiter conflict (for example, a seize).

- ▶ ***NONCONFLICTCALLSTACK**

Call stacks are captured for another conflict condition when there is not necessarily a conflicting holder.

- ▶ **Minimum duration (usecs)**

Minimum wait in order to produce a call stack for either ***CONFLICTCALLSTACK** or ***NONCONFLICTCALLSTACK**. Time is measured in microseconds (usecs).

DATATYPES - Data selection string

Similar to the thread selection string parameter, this parameter provides an alternate means to specify the Fixed data types and Dynamic data types parameters.

TRIGF - Conditional collection control

This option is used to conditionally gather data when certain performance thresholds have been met. A file must be provided to the engine that contains the conditional criteria selection string.

This conditional criteria selection string may contain direct field comparisons as well as functional comparisons. When making comparisons, the following operators must be used: **.GT.**, **.LT.**, **.EQ.**, **.NE.**, **.GE.**, **.LE.**

Direct field comparison:

Format: **fieldname .op. value**

Valid fields: Any fields in file QAPYJWTDE or QAPYJWPRC.

Example 6-2 Using the direct field comparisons

```
QTIME01 .GT. 200 .OR. QTIME05 .GT. 1000
```

Functional comparisons:

These comparisons involve a function applied to a field such as **RATE** or **PERCENT**.

Percent function

Format: **PERCENT(fieldname) .op. value**

Valid fields: Any wait bucket **TIME** values. Field names **QTIME01-QTIME32**

Example 6-3 Using the percent function

```
PERCENT(QTIME09) .GT. 50
```

Average function

Format: AVERAGE(fieldname1)(fieldname2) .op. value

Valid fields: This function requires a time/count pair, and both fields must be specified. The only pairs like this that are available at this time are the wait bucket fields QTIME01-QTIME32 and QCOUNT01-QCOUNT32.

Example: D-1 Using the average function

```
AVERAGE(QTIME01) (QCOUNT01) .LT. 35
```

Rate function

Format: RATE(fieldname).comparand.value

Valid fields: Fields QCOUNT01-CCOUNT32

Example: D-2 Using the rate function

```
RATE(QCOUNT08) .EQ. 25
```

File name

Name of the source file containing the conditional criteria selection string.

Library name

The library name where the conditional criteria file resides.

TRIGFMBR - Condition collection file member

Name of the conditional criteria file member.

ENDASPTHR - DB2 file ASP threshold

End the collection based on the ASP threshold or override with a percentage.

***ASPTHR (Default)**

Collection will end if the Auxiliary Storage Pool (ASP) threshold defined in the DST/SST Work with ASP threshold option is exceeded.

Percentage

Override the ASP threshold with another percentage. This change will not affect the current ASP setting and is valid only for this collection.

SETBLKMAP - Use WCHJOB block mappings

Job Watcher default block bucket mappings. For IBM representatives only.

TEXT - Text description

Description of the collection.

DTAAVAIL - DB2 data availability

This option controls how often records are written out to the database files during collection. The following choices are available:

***ITVEND (Default)**

The database files will be updated immediately after each interval is gathered.

***END**

The database files will not be updated after each interval. Data is blocked in large chunks and written out to the database files when the block of data is full or the Job Watch ends. This option can be used to greatly increase the speed at which the Job Watcher engine can sample data.

Note: Use this parameter when you do not need to look at the data in real-time mode (for example, if you save the data of one system and restore it to another system for analysis).

MINCPUDTAT - Minimum CPU statistics gathering time

This parameter determines how often CPU statistics are gathered. This value determines the number of seconds to wait before gathering the CPU information again.

Note: A separate file is created by the WCHJOB command called QAIDRJWCPU, which collects many different types of CPU statistics.

CPYJWCOL command

Use this command to copy a Job Watch. This action copies every member matching the Job Watch name from the *from* library to the *to* library. If you want to, the Job Watch can be renamed by setting the TOCOL parameter with a new name.

CPYJWCOL parameters

FROMCOL - From Job Watch Name

The name of the Job Watch to copy.

FROMLIB - From Job Watch Library Name

The library name of the Job Watch to copy.

TOCOL - To Job Watch Name

The name to give the Job Watch when copied to its new library.

TOLIB - To Job Watch Library Name

The name of the library to copy the Job Watch to. The library may be the same as the FROMLIB parameter as long as the TOCOL parameter differs from the FROMCOL parameter.

DLTJWCOL command

Use this command to delete a Job Watch from a user's library on a system. This action removes all members matching the Job Watch name from all QAPYJW* files found in the library specified.

The command also removes the record in file QAIDRJWRI where the MBR field value equals the Job Watch name. This single member file is used by the GUI to keep track of the Job Watches that have been created in the library.

DLTJWCOL parameters

WCHNAME - Job Watch name

The name of the Job Watch to delete. This value equals the member name used in the Job Watcher output files.

LIB - Job Watch library name

The library name from which the Job Watch will be deleted.

FTPJWCOL command

Use this command to send a Job Watch to another system. To perform this action, you need to provide the name of the Job Watch to be transferred along with information about the remote system you are sending the saved Job Watch to including the system name, library and file name.

This command saves the collection using command Save a Job Watch (SAVJWCOL) into a save file and then utilizes command FTPFILE in library QIDRGUI to perform the actual file transfer. The collection is not restored by this command. This must be done on the remote system by using command Restore a Job Watch (RSTJWCOL).

If a problem occurs while attempting to transfer the file to the remote system, a log file showing the content of the FTP session is saved to the file FTPLOG in library QIDRGUI.

FTPJWCOL parameters

COL - Job Watch name

The name of the Job Watch to send to the remote system.

LIB - Job Watch library name

The library name of the Job Watch to send to the remote system.

RMTSYSTEM - Remote system

The name of the remote system to transfer the collection to.

RMTLIB - Remote library

The name of the library on the remote system to transfer the save file containing the collection to.

RMTFILE - Remote file

The name of the save file to create on the remote system containing the collection.

USR - User name for remote system

The name of the user profile to use when making the connection to the remote system.

PWD - Password for remote system

The password to use for the user profile when making the connection to the remote system.

This command can also be run from the GUI by right-clicking on a Job Watcher collection and clicking on **Transfer to**. This displays the window shown in Figure D-1 on page 249. Fill in the

name of the system you are transferring the collection to. Fill in the name of the Remote library to transfer the save file containing the collection. Click **OK**.

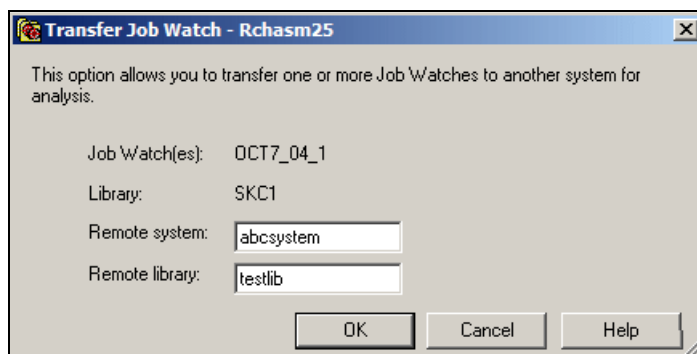


Figure D-1 Transfer Job Watch

RSTJWCOL command

Use this command to restore a Job Watch from a save file created by the Save a Job Watch (SAVJWCOL) command or indirectly using the FTP a Job Watch command (FTPJWCOL). The save file that is being restored must have been saved over a collection created on a system running i5/OS V5R3.

RSTJWCOL parameters

SAVFLIB - Save file library

The library name of the save file to restore.

SAVFNAME - Save file name

The name of the save file to restore.

SAVEDLIB - Saved library

The name of the library saved within the save file being restored. Use the Display Save File (DSPSAVF) command on the save file being restored to determine this library name.

TOLIB - Restore to library

The name of the library to restore the Job Watch to.

SAVJWCOL command

Use this command to save a Job Watch that was created on a system running i5/OS V5R3. The Job Watch is saved to a save file specified by the user. This save file must already exist. If it contains data, this data will be destroyed.

This command should be used along with the Restore a Job Watch command (RSTJWCOL) in order to restore the Job Watch.

SAVJWCOL parameters

COL - Job Watch name

The name of the Job Watch to save.

LIB - Job Watch library name

The library name of the Job Watch to save.

SAVFNAME - Save file name

The name of the save file to save the Job Watch into.

SAVFLIB - Save file library name

The library name of the save file to save the Job Watch into.

ADDPRDACS command

Use this command if you need to manually enter an access code for your system. This command can be found on an i5/OS V5R3 system in any of the iDoctor libraries that include QIDRWCH, QIDRPA, and QIDRGUI.

RTVSTKDTA command

When running a Job Watcher collection you can optionally specify to collect call stack information. The Job Watcher GUI call stack functions uses collected call stack information, but for a single job, thread, or task through V5R3.

Use this Retrieve Call Stack Data command to export the call stack information from an internally stored format to the database file called QAIDRJWSTK. The exported data residing within QAIDJWSTK records/rows is ready for user-written SQL queries. With QAIDJWSTK file data you can write queries that do things such as derive call stack statistics across all jobs/threads/tasks within the collection.

One example would be view all call stacks where program xxxxx occurred.

This readbook uses the RTVSTKDTA command and its output data in several places in the following chapters:

- ▶ Chapter 4, “Analysis example: traditional batch ILE RPG application” on page 117
- ▶ Chapter 5, “SQL, call stack, and journal analysis examples” on page 125
- ▶ Chapter 6, “Analysis example: Java application” on page 159

The RTVSTKDTA command parameters are described in full in this topic.

tRTVSTKDTA parameters

WCHMBR - Watch member name

The member name of the Job Watch collection.

WCHLIB - Watch library name

The library name of the Job Watch collection.

MODTYPE - Module name and VRM

iDoctor for iSeries version, release and modification level.

V5R3M0 is the only valid value for Job Watcher V5R3 data.

RCOUNT - Interval or QRECN (or *ALL)

Records subset option. Specify the interval number (right-justified in the 10-character formatted parameter space) or *ALL for all intervals.

Use *ALL unless you are only interested in the call stack for a single interval.

TCOUNT - Task count value (or *ALL)

The specific taskcount number (right-justified in the 20-character formatted parameter space) or *ALL for all jobs/threads/tasks collected.

Use *ALL unless you are only interested in the call stack for a single task count.

When you enter a single task count, say 38903, it must be entered as a 20-digit value with leading zeroes. For example, 000000000000000038903.

DEBUG - Debug/dump request

Leave this set to the default option of 0, unless instructed to enter a specific value by an IBM defect support personnel.

STKMBRNAME - Frame outfile member name

The name of the member in the output file QAIDRJWSTK for the resolved call stack data.

When you are expanding call stack data for the whole collection, use the Job Watch collection name.

Automatically submit a Job Watch

If you are having intermittent performance problems and not sure when the problem might occur, this Command Language (CL) source code is provided to automatically submit five Job Watches every two hours throughout the day. The source is provided as-is and may be modified to fit your specific needs.

The source creates five libraries called IBMJOBW1, IBMJOBW2, IBMJOBW3, IBMJOBW4, and IBMJOBW5. The collection data for each watch is stored in these libraries with a member name of Jmmddx, where mm = the current two-character month, dd = the current two-character day, and x = A, B, C, D, or E depending on the collection.

The Job Watch is set up with the following parameters:

- ▶ Data is collected for 2 hours or until the maximum data to collect reaches 4 GB.
- ▶ Data is collected over all jobs and tasks.
- ▶ Always collect the call stack data.
- ▶ Interval duration of 10 seconds.
- ▶ Always collect SQL and host variables.
- ▶ Data is not dumped to database files until the collection ends.

Example: D-3 Submitting a Job Watch automatically

PGM

DCL VAR(&MONTH) TYPE(*CHAR) LEN(2)

```

DCL      VAR(&DAY) TYPE(*CHAR) LEN(2)
DCL      VAR(&JW) TYPE(*CHAR) LEN(1) VALUE('J')
DCL      VAR(&MBR) TYPE(*CHAR) LEN(8)
DCL      VAR(&NBR) TYPE(*DEC) LEN(1 0) VALUE(1)
DCL      VAR(&CHAR) TYPE(*CHAR) LEN(1)
DCL      VAR(&LIB) TYPE(*CHAR) LEN(7) VALUE('IBMJOBW')
DCL      VAR(&LIB1) TYPE(*CHAR) LEN(8)

/* CHECK FOR PREVIOUS EXISTENCE OF LIBRARIES */
CHKOBJ   OBJ(IBMJOBW1) OBJTYPE(*LIB)
MONMSG   MSGID(CPF9801) EXEC(CRTLIB LIB(IBMJOBW1) +
    TEXT('IBM Job Watcher data library'))

CHKOBJ   OBJ(IBMJOBW2) OBJTYPE(*LIB)
MONMSG   MSGID(CPF9801) EXEC(CRTLIB LIB(IBMJOBW2) +
    TEXT('IBM Job Watcher data library'))

CHKOBJ   OBJ(IBMJOBW3) OBJTYPE(*LIB)
MONMSG   MSGID(CPF9801) EXEC(CRTLIB LIB(IBMJOBW3) +
    TEXT('IBM Job Watcher data library'))

CHKOBJ   OBJ(IBMJOBW4) OBJTYPE(*LIB)
MONMSG   MSGID(CPF9801) EXEC(CRTLIB LIB(IBMJOBW4) +
    TEXT('IBM Job Watcher data library'))

CHKOBJ   OBJ(IBMJOBW5) OBJTYPE(*LIB)
MONMSG   MSGID(CPF9801) EXEC(CRTLIB LIB(IBMJOBW5) +
    TEXT('IBM Job Watcher data library'))

/* SETUP SINGLE THREADED JOBQ */

CRTJOBQ   JOBQ(QGPL/QIDRJW1) TEXT('SINGLE THREAD JW +
    JOBQ')
MONMSG   MSGID(CPF3323)

ADDJOBQE  SBSQ(QIDRJW) JOBQ(QIDRJW1) SEQNBR(20)
MONMSG   MSGID(CPD1535 CPF1697)

/* RETRIEVE MONTH AND DAY */

RTVSYSVAL SYSVAL(QDAY) RTNVAR(&DAY)
RTVSYSVAL SYSVAL(QMONTH) RTNVAR(&MONTH)

/* SUBMIT JOB WATCHER USING THE FOLLOWING DATA OPTIONS: */
/* DASDDB MAX IS 4GB */
/* ALL JOBS AND TASKS */
/* CALL STACK ALWAYS */
/* 10 SEC. INTERVALS */
/* SQL & HOST VARIABLES ALWAYS */
/* JOB WATCHER DATA IS NOT DUMPED UNTIL THE COLLECTION ENDS */
/* COLLECTION RUNS FOR 2 HOURS */

CHGVAR    VAR(&CHAR) VALUE(&NBR)
CHGVAR    VAR(&MBR) VALUE(&JW *CAT &MONTH *CAT &DAY +
    *CAT 'A')
CHGVAR    VAR(&LIB1) VALUE(&LIB *CAT &CHAR)

SBMJOB    CMD(QIDRWCH/WCHJOB OUTLIBMBR(&LIB1 &MBR) +
    REPLACE(*YES) UNTIL(((DASDDB 4000)) +
    ((NBRSEC 7200))) INTDELAY(10) +

```



```

        JOBS(*ALLJOBS) TASKS(*ALL) +
        DATATYPEU((*CALLSTACK *ALWAYS) +
        (*SQLCURSTMT *ALWAYS) (*ACTGRPDETAIL +
        *ALWAYS)) TEXT('') DTAAVAIL(*END) +
        FRC1STINT(*YES)) JOB(QWCHJOB) +
        JOBD(QIDRWCH/QWCHJOB) JOBQ(QGPL/QIDRJW1) +
        RTGDTA('QWCHJOB') INLLIBL(*CURRENT)

CHGVAR  VAR(&NBR) VALUE(&NBR + 1)
CHGVAR  VAR(&CHAR) VALUE(&NBR)
CHGVAR  VAR(&MBR) VALUE(&JW *CAT &MONTH *CAT &DAY +
        *CAT 'B')
CHGVAR  VAR(&LIB1) VALUE(&LIB *CAT &CHAR)

SBMJOB  CMD(QIDRWCH/WCHJOB OUTLIBMBR(&LIB1 &MBR) +
        REPLACE(*YES) UNTIL(((DASDMB 4000)) +
        ((NBRSEC 7200))) INTDELAY(10) +
        JOBS(*ALLJOBS) TASKS(*ALL) +
        DATATYPEU((*CALLSTACK *ALWAYS) +
        (*SQLCURSTMT *ALWAYS) (*ACTGRPDETAIL +
        *ALWAYS)) TEXT('') DTAAVAIL(*END) +
        FRC1STINT(*YES)) JOB(QWCHJOB) +
        JOBD(QIDRWCH/QWCHJOB) JOBQ(QGPL/QIDRJW1) +
        RTGDTA('QWCHJOB') INLLIBL(*CURRENT)

CHGVAR  VAR(&NBR) VALUE(&NBR + 1)
CHGVAR  VAR(&CHAR) VALUE(&NBR)
CHGVAR  VAR(&MBR) VALUE(&JW *CAT &MONTH *CAT &DAY +
        *CAT 'C')
CHGVAR  VAR(&LIB1) VALUE(&LIB *CAT &CHAR)

SBMJOB  CMD(QIDRWCH/WCHJOB OUTLIBMBR(&LIB1 &MBR) +
        REPLACE(*YES) UNTIL(((DASDMB 4000)) +
        ((NBRSEC 7200))) INTDELAY(10) +
        JOBS(*ALLJOBS) TASKS(*ALL) +
        DATATYPEU((*CALLSTACK *ALWAYS) +
        (*SQLCURSTMT *ALWAYS) (*ACTGRPDETAIL +
        *ALWAYS)) TEXT('') DTAAVAIL(*END) +
        FRC1STINT(*YES)) JOB(QWCHJOB) +
        JOBD(QIDRWCH/QWCHJOB) JOBQ(QGPL/QIDRJW1) +
        RTGDTA('QWCHJOB') INLLIBL(*CURRENT)

CHGVAR  VAR(&NBR) VALUE(&NBR + 1)
CHGVAR  VAR(&CHAR) VALUE(&NBR)
CHGVAR  VAR(&MBR) VALUE(&JW *CAT &MONTH *CAT &DAY +
        *CAT 'D')
CHGVAR  VAR(&LIB1) VALUE(&LIB *CAT &CHAR)

SBMJOB  CMD(QIDRWCH/WCHJOB OUTLIBMBR(&LIB1 &MBR) +
        REPLACE(*YES) UNTIL(((DASDMB 4000)) +
        ((NBRSEC 7200))) INTDELAY(10) +
        JOBS(*ALLJOBS) TASKS(*ALL) +
        DATATYPEU((*CALLSTACK *ALWAYS) +
        (*SQLCURSTMT *ALWAYS) (*ACTGRPDETAIL +
        *ALWAYS)) TEXT('') DTAAVAIL(*END) +
        FRC1STINT(*YES)) JOB(QWCHJOB) +
        JOBD(QIDRWCH/QWCHJOB) JOBQ(QGPL/QIDRJW1) +
        RTGDTA('QWCHJOB') INLLIBL(*CURRENT)

CHGVAR  VAR(&NBR) VALUE(&NBR + 1)

```

```

CHGVAR      VAR(&CHAR) VALUE(&NBR)
CHGVAR      VAR(&MBR) VALUE(&JW *CAT &MONTH *CAT &DAY +
                        *CAT 'E')
CHGVAR      VAR(&LIB1) VALUE(&LIB *CAT &CHAR)

SBMJOB      CMD(QIDRWCH/WCHJOB OUTLIBMBR(&LIB1 &MBR) +
                REPLACE(*YES) UNTIL((( *DASDMB 4000)) +
                (( *NBRSEC 7200))) INTDELAY(10) +
                JOBS(*ALLJOBS) TASKS(*ALL) +
                DATATYPEU(*CALLSTACK *ALWAYS) +
                (*SQLCURSTMT *ALWAYS) (*ACTGRPDETAIL +
                *ALWAYS)) TEXT('') DTAAVAIL(*END) +
                FRC1STINT(*YES)) JOB(QWCHJOB) +
                JOBD(QIDRWCH/QWCHJOB) JOBQ(QGPL/QIDRJW1) +
                RTGDTA('QWCHJOB') INLLIBL(*CURRENT)

```

ENDPGM

Job Watcher advanced topics

This appendix highlights the features in Job Watcher that are not integrated into the server and client code at the time of writing this book.

The Job Watcher client and server levels used throughout this book were:

- ▶ Client build reference level C00367
- ▶ Server build reference level S00050

In Chapter 6, “Analysis example: Java application” on page 159, we also use iDoctor for iSeries Heap Analyzer. The Heap Analyzer client and server levels used were:

- ▶ Client build reference level C00368
- ▶ Server build reference level S00024

For each feature described, we clearly state whether it is available in any form at the current release or planned for a future release.

You must first understand more detail about the Job Watcher collection process before you can appreciate why some of the new functions are actually necessary.

Important: This appendix addresses several topics that indicate that changes may occur in a future Job Watcher release or update. For the latest information about Job Watcher capabilities and fixes, go to:

http://www-912.ibm.com/i_dir/idoctor.nsf/jwUpdateHistory.html

You can also reach the Update history page from:

<http://www.ibm.com/servers/eserver/support/series/>

Click the links for **Tools** and **Performance Analysis (iDoctor for iSeries)**. In the left navigation pane, expand **Job Watcher** and select **Update History**.

Collection specification tips

Interval size

If you suspect that a large number of task counts are active on your system, then you need to carefully decide what to set the collection interval to. If you suspect that your interval size is too small, then run the query in Example E-1 over the collection member in the QAPYJWTDE file to find out the maximum seconds that Job Watcher needed to finish its snapshot work.

Example: E-1 Query to find snapshot times and total active task counts

```
SELECT interval, count(distinct(taskcount)) as tot_taskcounts,  
       max(startod) as max_startod, min(startod) as min_strtod,  
       max(startod) - min(startod) as snapshot_secs  
FROM somejwlib/qapyjwtde  
GROUP BY interval  
ORDER BY 5 desc
```

The following data is from a customer system model 9406-890 32 processor system. This collection was run for two hours with a 5-second collection interval. The size of this Job Watcher collection was 3.1 GB.

Example: E-2 Snapshot times and total active task counts on a large, busy system

Interval number	TOT_TASKCOUNTS	MAX_STARTOD	MIN_STRTOD	SNAPSHOT_SECS
110	4,630	2004-08-18-11.16.55.564000	2004-08-18-11.12.59.790000	355.774000
273	1,704	2004-08-18-11.35.06.411000	2004-08-18-11.34.01.203000	105.208000
274	2,495	2004-08-18-11.35.46.672000	2004-08-18-11.35.09.269000	37.403000
1,046	1,934	2004-08-18-12.51.46.944000	2004-08-18-12.51.18.058000	28.886000
797	2,071	2004-08-18-12.28.20.727000	2004-08-18-12.27.53.451000	27.276000
349	1,592	2004-08-18-11.44.28.600000	2004-08-18-11.44.03.576000	25.024000
406	2,257	2004-08-18-11.49.58.713000	2004-08-18-11.49.33.978000	24.735000
705	1,763	2004-08-18-12.19.07.442000	2004-08-18-12.18.43.736000	23.706000
243	1,770	2004-08-18-11.31.11.265000	2004-08-18-11.30.47.876000	23.389000
410	2,234	2004-08-18-11.50.47.800000	2004-08-18-11.50.24.817000	22.983000
131	1,254	2004-08-18-11.20.26.668000	2004-08-18-11.20.03.725000	22.943000
275	2,696	2004-08-18-11.36.47.680000	2004-08-18-11.36.24.841000	22.839000
967	1,566	2004-08-18-12.44.17.027000	2004-08-18-12.43.54.759000	22.268000
1	828	2004-08-18-11.03.40.378000	2004-08-18-11.03.20.450000	19.928000
788	1,884	2004-08-18-12.27.00.465000	2004-08-18-12.26.41.705000	18.760000
557	1,625	2004-08-18-12.05.22.895000	2004-08-18-12.05.04.340000	18.555000

The above collection data revealed that it took from 1 second to 15 seconds for Job Watcher to complete both data capture and delta computations, as there were 25,000+ task counts of which approximately 1,500 used CPU in an average interval. When the number of active task counts suddenly increases, say to 4,500+ from 1,500, then the time to complete the snapshot collection and delta calculations can be very high, for example 355+ seconds shown for interval 110 and 105+ seconds for interval 273.

Ensure job names are captured for all jobs on the system

Job Watcher enables you to select **Collect data for idle threads/tasks on first interval**; see Figure E-1 on page 257 for how to do this. If you select this option, records for all task counts on the system will be written to the QAPYJWTDE file for interval 1. Of course if you do not select this option, then only records for task counts that used > 0 CPU will be written to the QAPYJWTDE file. Using this option ensures that you will always be able to identify wait object holder job names. If you do not specify this option and a job stays in a long wait for every snapshot in the collection, then you will not know the associated job name.

If you specified **Collect data for idle threads in the first interval** as shown in Figure E-1, then at the end of the second snapshot, the data for all task counts on the system is written to the QAPYJWTDE file; this causes additional overhead on busy systems.

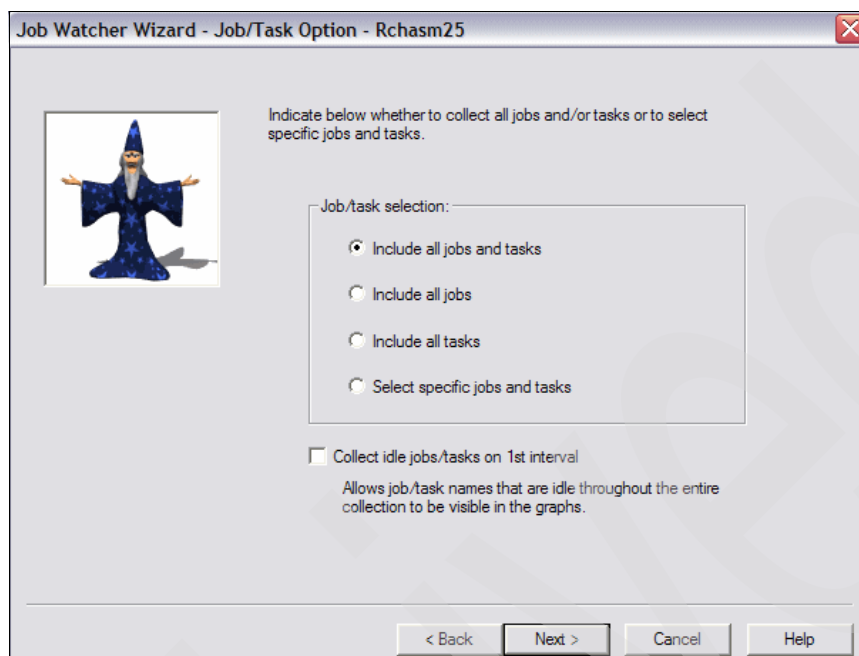


Figure E-1 Collect idle job/thread data for first interval

Collect data as fast as possible?

This collection option is intended for use over a few jobs and LIC tasks. You should *not* use this collection option for all jobs and LIC tasks unless directed to do so by IBM Service. Of course, this does not apply if you are only collecting data for one or two jobs in the system.

The Job Watcher collection process in detail

Figure E-2 on page 258 shows an example of how an individual task count may behave from the perspective of the Job Watcher collector. Assume that this is a task count for the primary thread in a job and that this job does not have any secondary threads.

This section contains a detailed description of how Job Watcher captures run and wait data for a task count for i5/OS V5R3.

Assume that the parameters that are specified at collection time ensure that this task count will always be included in the collection (for example, all tasks and all threads).

The complexities of explaining how the collection process works for a task count in a job running under an active user profile that is continually changing, where the collection parameters specified jobs running under only one of the user profiles, are outside the scope of this publication. This same exclusion applies to a task count in a job that runs in different subsystems, where not all of those subsystems are included in the collection.

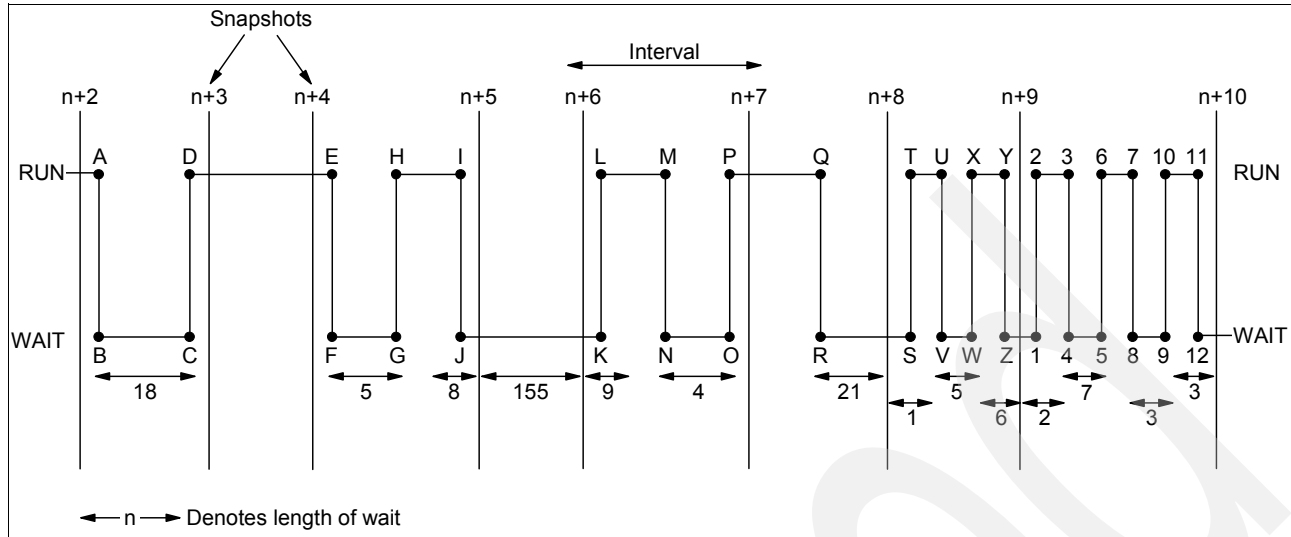


Figure E-2 Collection process for one task count

Snapshot

A sample of data taken at a point in time by the Job Watcher for eligible task counts. In our example there are nine snapshots, shown as $n+2$, $n+3$... $n+10$.

Interval n occurs at the end of snapshot $(n+1)$ when the delta statistics are computed between snapshot $(n+1)$ and snapshot n . In our example, eight intervals are shown.

Delta statistics are calculated from snapshot 2 onward in a collection.

What are the main collection database files?

The two main files in every collection are:

- ▶ QAPYJWTDE: the TDE file
- ▶ QAPYJWSTS: the status file

The collection options you specify determine whether data will be written to many of the Job Watcher QAPYJW* or QAIDRJW* files in the collection library. Refer to Appendix B, "Database files created by Job Watcher" on page 199 for a list of the files created by the collection process.

A record is always written to the QAPYJWSTS file for every task count (in the collection) in every interval. When a task count has not used any CPU in an interval, there is no QAPYJWTDE record for that interval, so no wait buckets are stored. The information about the wait in progress is kept in the QAPYJWSTS file.

A record is written to the QAPYJWTDE file for an interval and task count only if any one of the following is true:

1. It is the first interval and **Collect data for idle jobs/threads at first interval** was specified.
2. The task count is a secondary thread in a job and that thread used > 0 CPU in the interval.
3. The task count is a primary thread in a job and one or more of the secondary threads in that job used > 0 CPU in the interval.
4. The task count is a primary thread in a job and that primary thread used > 0 CPU in the interval and data for this interval and task count has not already been written out because of item 3 in this list.

For each interval in our example shown in Figure E-2 on page 258, we discuss which of these files has a record written to it, as listed in Table E-1.

Table E-1 When is data written to the Job Watcher QAPYJWTDE and QAPYJWSTS master file?

Snapshot: interval	QAPYJWSTS (record status)	QAPYJWTDE (record written)
n+3: Interval n+2	(A) task used some CPU	(yes)
n+4: Interval n+3	(A)	(yes)
n+5: Interval n+4	(A)	(yes)
n+6: Interval n+5	(I) task used no CPU	(no) the task was in wait for the entire interval
n+7: Interval n+6	(A)	(yes)
n+8: Interval n+7	(A)	(yes)
n+9: Interval n+8	(A)	(yes)
n+10: Interval n+9	(A)	(yes)

What data is collected?

The system wait buckets are stored in the TDE for each task count. There are 32 buckets per task count. Refer to Chapter 2, “Overview of job waits and iDoctor for iSeries Job Watcher” on page 21 for details. Each system wait bucket has two parts: cumulative count and cumulative time. See 2.4, “The mysteries of waiting” on page 26 through 2.8.2, “Job Watcher wait points (ENUMs) and wait buckets” on page 30 for full details of wait types, wait groupings, and wait grouping to bucket number assignments.

For simplicity, assume that all waits shown on the wait line in Figure E-2 on page 258 are seize waits. This means that we need to discuss only two of the buckets in order to understand the Job Watcher collection process:

- ▶ Bucket 01 - dispatched to CPU. See 2.9.2, “Bucket 1: Dispatched Time (previously referred to as CPU)” on page 33 for detailed description.
- ▶ Bucket 15 - seize waits.

Job Watcher takes snapshots of the task counts that are eligible for collection. If you specify that all jobs and tasks be included, then this is every task count on the system.

What happens at snapshot time?

At the first and second snapshots of any collection, information for all task counts on the system is gathered; after the second snapshot the deltas are computed and only data for task counts that used > 0 CPU between snapshots is written to the QAPYJWTDE file; this is the data for the first interval. A record for every task count in the collection is written to the QPAYJWSTS file for every interval.

Of course, if you collect the call stack and other detailed information (such as SQL data or communications data) at each interval then this causes even more overhead. The Job Watcher then computes delta information between snapshots for each task count and stores this information in the QAPYJWTDE file, in your chosen collection member name.

When the first snapshot occurs, Job Watcher determines the task counts that are eligible for collection, and system wait buckets are captured for these task counts. These system wait buckets are saved as Job Watcher wait buckets. At the next snapshot, that preceding interval’s data must be saved. This is done by getting the system wait buckets for eligible task

counts, calculating the deltas between the new system wait buckets and the Job Watcher wait buckets, and saving the delta bucket values in the QAPYJWTD file.

Table E-2 shows how Job Watcher gets the run and wait data for the task count in our example shown in Figure E-2 on page 258.

Note: The system LIC wait buckets are always present in the system for every task count. They are always updated by SLIC, irrespective of whether Job Watcher exists on the system. In Table E-2, the term LIC wait buckets is used to represent the system LIC wait buckets.

JW is used to represent Job Watcher throughout the following table.

Job Watcher does not update the system LIC wait buckets. It updates a saved copy of them, which are called the *copy-LIC wait buckets* in this table.

Table E-2 Job Watcher collection detailed explanation

Time line	Collection detail
Snapshot at n+1	Assume that the task count used > 0 CPU between snapshots n and (n+1) LIC wait buckets are harvested as CopyLIC (n+1) buckets and subsequently saved as JW (n+1) buckets because the task count used > 0 CPU between snapshot n and snapshot (n+1). Job Watcher knows the times when the task count last changed state and when it was last running.
Snapshot at n+2	The task count is in run state (active, dispatched for CPU) at snapshot time. This state is known as being switched in (or SWIN for short) by LIC. The LIC wait buckets are harvested and saved as CopyLIC (n+2) buckets. Delta values are calculated between the CopyLIC (n+2) buckets and the Job Watcher (n+1) buckets. The delta values are written in the record to the QAPYJWTD file as the task count used > 0 CPU. Now assume CopyLIC (n+2) bucket 15 has values: count = 3 and time = 83 Write record to QAPYJWSTS file. Save CopyLIC (n+2) buckets as Job Watcher (n+2) buckets. Job Watcher (n+2) bucket 15 has values: count = 3 and time = 83.
A to B	Task count transitions from being active at A to waiting at B. This state change from being active to waiting is known in LIC as changing from being switched in (SWIN) to being switched out on queue (SWOQ).
C	The wait ends (is satisfied) at C. It lasted 18 units of time.
C to D	Task count transitions from waiting at B to active at D. This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ for short) to being switched in (SWIN for short).
D	LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 and 18 is added to the time. LIC wait bucket 15 now has values: count = 4 and time = 101. The LIC wait buckets are not available to Job Watcher until the next snapshot.

Time line	Collection detail
Snapshot at n+3	<p>The task count is in run state (active, dispatched for CPU) at snapshot time. This state is known as being switched in (or SWIN for short) by LIC.</p> <p>The LIC wait buckets are harvested and saved as CopyLIC (n+3) buckets. Job Watcher knows when the last state change for this task count (SWIN to SWOQ) occurred. Job Watcher then adds time D to CopyLIC (n+3) bucket 1.</p> <p>Delta values are calculated between the CopyLIC (n+3) buckets and the Job Watcher (n+2) buckets. The delta values (bucket 15: count 1 time 18) and the delta dispatched for CPU time (bucket 1) are written in the record to the QAPYJWTDE file as the task count used > 0 CPU.</p> <p>Write record to QAPYJWSTS file. Save CopyLIC (n+3) buckets as Job Watcher (n+3) buckets. Job Watcher (n+3) bucket 15 has values: count = 4 and time = 101.</p>
Snapshot at n+4	<p>The task count is in run state (active, dispatched for CPU) at snapshot time. This state is known as being switched in (or SWIN for short) by LIC.</p> <p>Job Watcher detects that there has been no state change since the last snapshot at (n+3) and it knows the last switched in time (SWIN); therefore the LIC wait buckets do not have to be harvested. CopyLIC (n+4) buckets are set to CopyLIC (n+3) buckets. Time (n+3) to (n+4) is added to CopyLIC (n+3) bucket 1.</p> <p>Delta values are calculated between the CopyLIC (n+4) buckets and the Job Watcher (n+3) buckets. Now the CPU time in the delta for bucket 1 is actually the time that occurred within interval n+3 to n+4. This is why the D to (n+3) addition was made to CopyLIC (N+4) bucket 1.</p> <p>Write record to QAPYJWTDE file as > 0 CPU was used since last snapshot. Write record to QAPYJWSTS file Save CopyLIC (n+4) buckets as Job Watcher (n+4) buckets Job Watcher (n+4) bucket 15 remains with values: count = 4 and time = 101.</p>
E to F	<p>Task count transitions from being active at E to waiting at F.</p> <p>This state change from being active to waiting is known in LIC as changing from being switched in (SWIN) to being switched out on queue (SWOQ).</p>
G	<p>The wait ends (is satisfied) at G. The wait lasted 5 units of time.</p>
G to H	<p>Task count transitions from waiting at G to active at H</p> <p>This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).</p>
H	<p>LIC wait buckets are updated for this task count by SLIC.</p> <p>The count in LIC wait bucket 15 is increased by 1 and 5 is added to the time.</p> <p>LIC wait bucket 15 now has values: count = 5 and time = 106</p> <p>The LIC wait buckets are not available to Job Watcher until the next snapshot.</p>
H to I	<p>Some CPU is used. The task count is switched in (SWIN) at H and is switched out on queue (SWOQ) at I as it goes into the wait.</p> <p>LIC wait bucket 1 is not incremented at this time (it will next be incremented at L as LIC wait buckets are only updated by SLIC when the task count goes to the switched in (SWIN) state.</p>
I to J	<p>Task count transitions from being active at I to waiting at J.</p> <p>This state change is known in LIC as changed from being switched in (SWIN) to being switched out on queue (SWOQ).</p>
J to n+5	<p>A wait of 8 seconds has occurred. The system wait buckets are not updated as this wait is still in progress at time n+5.</p>

Time line	Collection detail
Snapshot at n+5	<p>The task count is in wait state at snapshot time. This state is known as being switched out on queue (or SWOQ) by LIC. Job Watcher knows when the task count state changed from RUNNING (switched in or SWIN for short) to WAITING (called switched out on queue or SWOQ for short); it also knows when the task count was last switched in (SWIN).</p> <p>The LIC wait buckets are harvested as CopyLIC (n+5) buckets. CopyLIC (n+5) wait bucket 15 has values: count = 5, time = 106 (last updated at H). Time H to I is added to CopyLIC (n+5) bucket 1. The count in CopyLIC (n+5) bucket 15 is increased by 1 and 8 is added to the time. CopyLIC (n+5) bucket 15 now has values: count = 6 and time = 114.</p> <p>Delta values are calculated between the CopyLIC (n+5) buckets and the Job Watcher (n+4) buckets. The delta values (bucket 15: count 2 time 13) are written in the record to the QAPYJWTD file as the task count used > 0 CPU.</p> <p>Write record to QAPYJWSTS file. Save CopyLIC (n+5) buckets as Job Watcher (n+5) buckets. Job Watcher (n+5) bucket 15 has values: count = 6 and time = 114.</p>
Snapshot at n+6	<p>The task count is in wait state at snapshot time. This is "switched out on queue" (or SWOQ) by LIC Job Watcher detects that there has been no state change since the last snapshot at (n+5) therefore the system wait buckets do not have to be harvested. Job Watcher knows the last state change was from active to wait (SWIN to SWOQ) and when it occurred. It also knows when the task count last used CPU (last switched in time or SWIN) and that this was before (n+5). Therefore it knows that no CPU was used in the last interval. CopyLIC (n+6) buckets are set to CopyLIC (n+5) buckets.</p> <p>Since the last state change was to waiting and no CPU was used in the last interval, the time (n+5) to (n+6) of 155 is added to CopyLIC (n+6) bucket 15 but the count is not incremented because the wait has not yet been satisfied. CopyLIC (n+5) bucket 15 now has values: count = 6 and time = 269.</p> <p>Write record to QAPYJWSTS file with current wait type and time of 269 (cumulative value as from system wait bucket corresponding to current wait type i.e. 15). Save CopyLIC (n+6) buckets as Job Watcher (n+6) buckets. Job Watcher (n+6) bucket 15 has values: count = 6 and time = 269.</p>
K to L	<p>Task count transitions from waiting at K to active at L. This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).</p>
L	<p>LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 because the wait has now been ended and 172 is added to the time. LIC wait bucket 15 now has values: count = 6 and time = 278. LIC wait buckets are not available to Job Watcher until the next snapshot.</p>
L to M	<p>Some CPU is used. The task count is switched in (SWIN) at L and is switched out on queue (SWOQ) at M as it goes into the wait. LIC wait bucket 1 is not incremented at this time (it will next be incremented at L as LIC wait buckets are only updated by SLIC when the task count goes to the switched in (SWIN) state.</p>
M to N	<p>Task count transitions from being active at M to waiting at N. This state change is known in LIC as changed from being switched in (SWIN) to being switched out on queue (SWOQ).</p>

Time line	Collection detail
O	The wait ends (is satisfied) at O. It lasted 4 units of time.
O to P	Task count transitions from waiting at O to active at P This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).
P	LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 and 4 is added to the time. LIC wait bucket 15 now has values: count = 7 and time = 282 LIC wait buckets are not available to Job Watcher until the next snapshot.
Snapshot at n+7	The task count is in run state (active, dispatched for CPU) at snapshot time. This state is known as being switched in (or SWIN for short) by LIC. The LIC wait buckets are harvested as CopyLIC (n+7) buckets. Job Watcher knows when the last state change for this task count (SWIN to SWOQ) occurred at P. Job Watcher then adds time P to (n+7) to CopyLIC (n+7) bucket 1. Delta values are calculated between the CopyLIC (n+7) buckets and the Job Watcher (n+6) buckets. The delta values (bucket 15: count 1 time 13) and the delta dispatched for CPU time (bucket 1) are written in the record to the QAPYJWTD file as the task count used > 0 CPU. The actual record written to the QAPYJWTD file says current/last wait time is 4 and total wait time for the interval is 13 - note that the 4 is included in the 13 as the wait was not in progress at snapshot time, so the 13 is the total time of all completed waits. Write record to QAPYJWSTS file. Save system wait buckets for (n+7) as Job Watcher wait buckets for (n+7). Job Watcher wait bucket 15 has values: count = 7 and time = 282.
Q to R	Task count transitions from being active at Q to waiting at R. This state change from being active to waiting is known in LIC as changing from being switched in (SWIN) to being switched out on queue (SWOQ).
Snapshot at n+8	The task count is in wait state at snapshot time. This state is known as being switched out on queue (or SWOQ) by LIC. Job Watcher knows when the task count changed from state RUNNING (switched in or SWIN for short) to WAITING (called switched out on queue or SWOQ for short); it also knows when the task count was last switched in (SWIN). The LIC wait buckets are harvested and saved as CopyLIC (n+8) buckets. CopyLIC (n+8) bucket 15 has values: count = 7 time = 282 (last updated at P). Time n+7 to Q is added to CopyLIC (n+8) bucket 1. The count in CopyLIC (n+8) bucket 15 is increased by 1 and 21 is added to the time. CopyLIC (n+8) bucket 15 now has values: count = 8 and time = 303 Delta values are calculated between the CopyLIC (n+8) buckets and the Job Watcher (n+7) buckets. The delta values (bucket 15: count 1 time 21) are written in the record to the QAPYJWTD file as the task count used > 0 CPU. Write record to QAPYJWSTS file. Save CopyLIC (n+8) buckets as Job Watcher (n+8) buckets. Job Watcher (n+7) bucket 15 has values: count = 8 and time = 303.
S to T	Task count transitions from waiting at S to active at T. This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).

Time line	Collection detail
T	LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 and 22 is added to the time. LIC wait bucket 15 now has values: count = 8 and time = 304. LIC wait buckets are not available to Job Watcher until the next snapshot.
T to U	Some CPU is used. The task count is switched in (SWIN) at T and is switched out on queue (SWOQ) at U as it goes into the wait. LIC wait bucket 1 is not incremented at this time (it will next be incremented at X as LIC wait buckets are only updated by SLIC when the task count goes to the switched in (SWIN) state.
U to V	Task count transitions from being active at U to waiting at V. This state change is known in LIC as changed from being switched in (SWIN) to being switched out on queue (SWOQ).
W	The wait ends (is satisfied) at W. It lasted 5 units of time.
W to X	Task count transitions from waiting at W to active at X. This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).
X	LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 and 5 is added to the time. LIC wait bucket 15 now has values: count = 9 and time = 309. LIC wait buckets are not available to Job Watcher until the next snapshot.
X to Y	Some CPU is used. The task count is switched in (SWIN) at X and is switched out on queue (SWOQ) at Y as it goes into the wait. LIC wait bucket 1 is not incremented at this time (it will next be incremented at 2 as LIC wait buckets are only updated by SLIC when the task count goes to the switched in (SWIN) state.
Y to Z	Task count transitions from being active at Y to waiting at Z. This state change is known in LIC as changed from being switched in (SWIN) to being switched out on queue (SWOQ).
Z to n+9	A wait of 6 seconds has occurred. LIC wait buckets are not updated as this wait is still in progress at time n+9.
Snapshot at n+9	The task count is in wait state at snapshot time. This state is known as being switched out on queue (or SWOQ) by LIC Job Watcher knows when the task count changed from state RUNNING (switched in or SWIN for short) to WAITING (called switched out on queue or SWOQ for short); it also knows when the task count was last switched in (SWIN). LIC wait buckets are harvested and saved as CopyLIC (n+9) buckets. CopyLIC (n+9) bucket 15 has values: count = 9 time = 309 (last updated at X). Time X to Y is added to CopyLIC (n+9) bucket 1. The count in CopyLIC (n+9) bucket 15 is increased by 1 and 6 is added to the time. CopyLIC (n+9) bucket 15 now has values: count = 10 and time = 315 Delta values are calculated between the CopyLIC (n+9) buckets and the Job Watcher (n+8) buckets. The delta values (bucket 15: count 2 time 12) are written in the record to the QAPYJWTD file as the task count used > 0 CPU. Write record to QAPYJWSTS file Save CopyLIC (n+9) buckets as Job Watcher (n+9) buckets Job Watcher (n+9) bucket 15 has values: count = 10 and time = 315

Time line	Collection detail
1 to 2	Task count transitions from waiting at 1 to active at 2. This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).
2	LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 and 8 is added to the time. LIC wait bucket 15 now has values: count = 10 and time = 317. LIC wait buckets are not available to Job Watcher until the next snapshot.
2 to 3	Some CPU is used. The task count is switched in (SWIN) at 2 and is switched out on queue (SWOQ) at 3 as it goes into the wait. LIC wait bucket 1 is not incremented at this time (it will next be incremented at 6) as system wait buckets are only updated by SLIC when the task count goes to the switched in (SWIN) state.
3 to 4	Task count transitions from being active at 3 to waiting at 4. This state change is known in LIC as changed from being switched in (SWIN) to being switched out on queue (SWOQ).
5	The wait ends (is satisfied) at 5. It lasted 7 units of time.
5 to 6	Task count transitions from waiting at 5 to active at 6. This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).
6	LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 and 7 is added to the time. LIC wait bucket 15 now has values: count = 11 and time = 324. LIC wait buckets are not available to Job Watcher until the next snapshot.
6 to 7	Some CPU is used. The task count is switched in (SWIN) at 7 and is switched out on queue (SWOQ) at 7 as it goes into the wait. LIC wait bucket 1 is not incremented at this time (it will next be incremented at 10) as LIC wait buckets are only updated by SLIC when the task count goes to the switched in (SWIN) state.
7 to 8	Task count transitions from being active at 7 to waiting at 8. This state change is known in LIC as changed from being switched in (SWIN) to being switched out on queue (SWOQ).
9	The wait ends (is satisfied) at 9. It lasted 3 units of time.
9 to 10	Task count transitions from waiting at 9 to active at 10. This state change from waiting to being dispatched to CPU is known in LIC as changing from being switched out on queue (SWOQ) to being switched in (SWIN).
10	LIC wait buckets are updated for this task count by SLIC. The count in LIC wait bucket 15 is increased by 1 and 3 is added to the time. LIC wait bucket 15 now has values: count = 12 and time = 327. LIC wait buckets are not available to Job Watcher until the next snapshot.
10 to 11	Some CPU is used. The task count is switched in (SWIN) at 10 and is switched out on queue (SWOQ) at 11 as it goes into the wait. LIC wait bucket 1 is not incremented at this time (it will next be incremented after n+10) as LIC wait buckets are only updated by SLIC when the task count goes to the switched in (SWIN) state.
11 to 12	Task count transitions from being active at 11 to waiting at 12. This state change is known in LIC as changed from being switched in (SWIN) to being switched out on queue (SWOQ).

Time line	Collection detail
12 to n+10	A wait of 3 seconds has occurred. LICwait buckets are not updated as this wait is still in progress at time n+10.
Snapshot at n+10	<p>The task count is in wait state at snapshot time. This state is known as being switched out on queue (or SWOQ) by LIC Job Watcher knows when the task count changed from state RUNNING (switched in or SWIN for short) to WAITING (called switched out on queue or SWOQ for short); it also knows when the task count was last switched in (SWIN).</p> <p>LIC wait buckets at (n+10) are harvested and saved as CopyLIC (n+10) buckets. CopyLIC (N+10) bucket 15 has values: count = 12 time = 327 (last updated at 10). Time from 10 to 11 is added to CopyLIC (N+10) bucket 1. The count in CopyLIC (n+10) bucket 15 is increased by 1 and 3 is added to the time. CopyLIC (n+10) bucket 15 now has values: count = 13 and time = 330.</p> <p>Delta values are calculated between the CopyLIC (n+10) buckets and the Job Watcher (n+9) buckets. The delta values (bucket 15: count 3 time 9) are written in the record to the QAPYJWTDE file as the task count used > 0 CPU.</p> <p>Write record to QAPYJWSTS file. Save CopyLIC (n+10) buckets as Job Watcher (n+10) buckets. Job Watcher (n+10) bucket 15 has values: count = 13 and time = 330.</p>

Task count state changes

You are now familiar with the states of:

- ▶ Switched In (SWIN): when a task count is dispatched to a CPU (running)
- ▶ Switched Out Queue (SWOQ): when a task count goes into a wait state

These terms are used in another iDoctor for iSeries tool, the PEX Task Switch Trace. Figure E-3 depicts this switch in and switch out process as discussed in Table E-2 on page 260, for Figure E-2 on page 258. See also the description following Figure E-3.

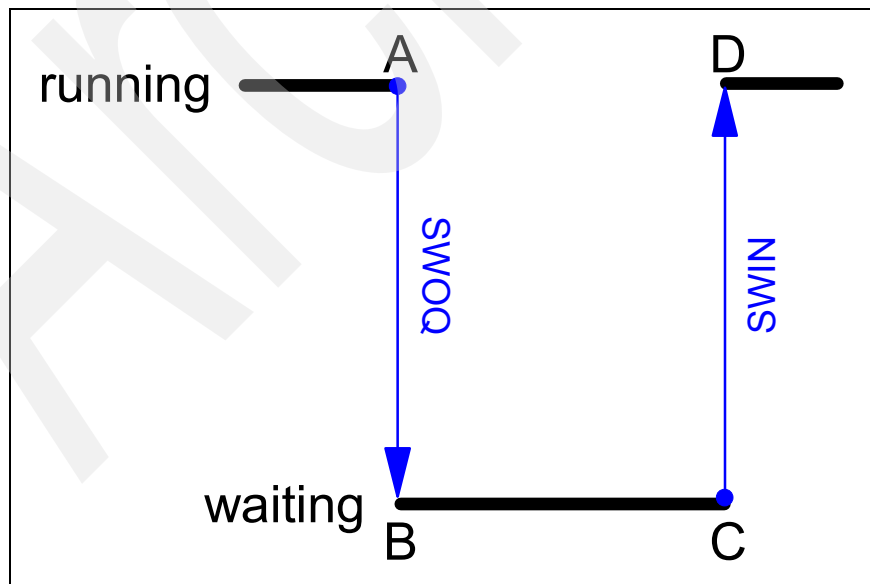


Figure E-3 Task count (switch) out and in

What happens when the CPU is not readily available?

You may be wondering what happens when the CPU is not instantly available and so a task count cannot immediately go into the switched in (SWIN) state. Actually, there is a transitional state between waiting and running that is available to run and needs a processor to run on; this state is known in LIC as Switched Available for Dispatch (SWAFD for short). A task count actually goes to SWAFD before it can go into the SWIN state. The difference between being available to use a CPU and being assigned a CPU is CPU queueing time; therefore CPU queueing time is the time from the SWAFD state to the SWIN state.

We did not show the SWAFD states in our example in Figure E-2 on page 258, as it was not necessary for explaining how Job Watcher captures data and when data is written to Job Watcher files. However, in reality, a task count would go to the SWAFD state before it gets assigned a CPU and goes to the SWIN state.

Waits that span multiple intervals

In our example (Figure E-2 on page 258), a seize wait was in progress for the whole interval from (n+5) to (n+6). This wait started before snapshot (n+5) and ended after snapshot (n+6).

The Job Watcher graphing support uses the QAPYJWTDE file as input.

There is a record in the QAPYJWTDE file for our task count for every interval except interval (n+5) to (n+6).

In the case of our example taskcount, which is the only taskcount in a job, the Run/wait signature for a job graph will show a blank for the entire interval (n+5) to (n+6). This is accurate from the perspective of the QAPYJWTDE file, but does not correctly represent what happened in interval (n+5) to (n+6) where the entire time was spent in seize wait. Also the Collection overview time signature graph will underrepresent the seize waits in this example.

An interval that shows up in Job Watcher Job run/wait signature graphs as all blank, but during which there was an ongoing wait, is known as a *blank interval*.

The QAPYJWSTS file contains a record for every task count for every interval. The wait time in this file is cumulative for an ongoing wait.

Figure E-4 on page 268 shows an example of what the graph of a Job run/wait signature looks like when there are blank intervals.

Notice the blank intervals at 147, 158, 159, and 178. Recall that this graph is created using the data in the QAPYJWTDE file, and that the QAPYJWTDE file does not have a record for intervals where the task count *spent the entire interval in a wait state*.

When a graph of job waits by interval has these blank intervals and either side of the blank interval there were lots of waits, it is likely that the blank interval contains only wait time. In this case you can guess that the job spent intervals 147, 158, 159, and 178 entirely in seize waits as these are the major wait type in this task count.

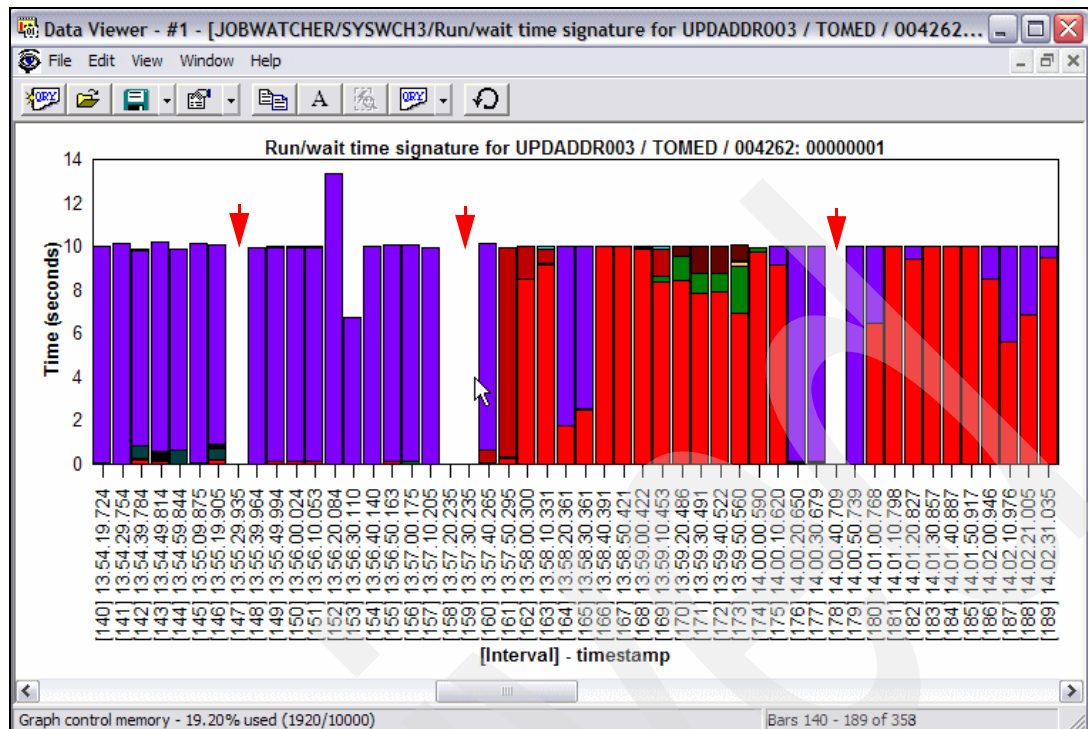


Figure E-4 Job run/wait signature with blank intervals

However, if there were many different types of wait occurring in the intervals surrounding a blank interval, then it may be difficult to guess the wait type that spanned the blank interval.

Check the QAPYJWTDE file

The blank intervals were 147, 158, 159, and 178. We need to investigate the intervals immediately preceding the blank intervals for current (in progress or ongoing) waits. The intervals of interest are 146, 157, and 177. We query the QAPYJWTDE file to see what the record had for that job and interval.

The SQL shown in Example E-3 runs the query shown in Example E-4.

Example: E-3 SQL for looking around the blank intervals in the QAPYJWTDE file

```
SELECT interval, blockbckt,
       (dec(currwtdur,9,0)) as currwait_microsec, woobjnam, woobjtyp,
       dec(qcount15,5,0) as bucket15_count, dec(qtime15,9,0) as
       bucket15_microsec
FROM jobwatcher/qapyjwtde
WHERE (interval >= 146 and interval <= 160 or
       interval >= 177 and interval <= 179 ) and taskcount = 1079
ORDER BY interval
```

We deliberately shortened some of the numeric output fields, using SQL casting via DEC(), to produce output within 132 characters.

Example: E-4 Looking around the blank intervals in the QAPYJWTDE file

Interval number	Current or last blocking bucket	CURRWAIT_MICROSEC	Wait object name	Wait object object type in hex	BUCKET15_COUNT	BUCKET15_MICROSEC
146	15	2,912,253	TESTBASEPFTSTBASEPF	0B90	22	9,166,903
148	15	172,609	TESTBASEPFTSTBASEPF	0B90	5	9,923,415

149	15	4,703,317	TESTBASEPFTSTBASEPF	0B90	18	9,826,975
150	15	9,617,923	TESTBASEPFTSTBASEPF	0B90	8	9,832,876
151	15	2,851,757	TESTBASEPFTSTBASEPF	0B90	18	9,825,700
152	15	177,016	TESTBASEPFTSTBASEPF	0B90	8	13,344,390
153	15	6,557,656	TESTBASEPFTSTBASEPF	0B90	1	6,726,905
154	15	2,784,132	TESTBASEPFTSTBASEPF	0B90	3	10,011,271
155	15	113,359	TESTBASEPFTSTBASEPF	0B90	11	9,953,659
156	15	3,184,844	TESTBASEPFTSTBASEPF	0B90	23	9,949,001
157	15	4,635,890	TESTBASEPFTSTBASEPF	0B90	2	9,936,060
160	11	6,979		0000	4	9,451,744
177	15	73,937	TESTBASEPFTSTBASEPF	0B90	3	9,923,307
179	15	2,033,771	TESTBASEPFTSTBASEPF	0B90	1	10,026,839

Analysis of this query reveals:

- Interval 146** The current block bucket is 15, which is seize contention. The task count was in a seize for 2,912,253 microseconds at interval end; this seize was on object TESTBASEPF of type physical file (0B90). Specifically, this task count was in a continuing seize wait when the end interval snapshot occurred.
- This task count spent 9,166,903 microseconds in seize contention wait in this interval. There were 22 individual seizes comprising this interval seize time. The ongoing seize wait of 2,912,253 microseconds is included in the interval seize wait total of 9,166,903.
- Interval 157** The current block bucket is seize contention (15). A 4,635,890 microsecond seize wait was in progress at interval end. Total seize wait time was 9,936,060, which includes the in-progress seize at interval end.
- Interval 177** The current block bucket is seize contention (15). A 73,937-microsecond seize wait was in progress at interval end. Total seize wait time was 9,923,307, which includes the in-progress seize at interval end.

Now we can conclude that for each of these blank intervals the task count was in seize wait time. Since the in-progress wait before the interval was a seize, and no CPU was used during the intervals, we know that this must be the type of wait that spanned the intervals as it is not possible to transition from one type of wait to another type of wait without using some CPU.

Check the QAPYJWSTS file

There is a record for every task count for every interval in this file.

There are no blank intervals in the QAPYJWSTS file, so we must check the interval numbers that were not present in the QAPYJWSTDE file (shown as blank intervals on the Job run/wait signature graph). The intervals of interest are 147, 158, 159, and 178 in the QPAYJWSTS file. The SQL we used is shown in Example E-5.

Example: E-5 SQL for blank interval investigation: check the QAPYJWSTS file

```
SELECT interval, tdestatus, curwait, curwaitd
FROM jobwatcher/qapyjwsts
WHERE (interval >= 146 and interval <= 160 or
interval >= 177 and interval <= 179 ) and taskcount = 1079
ORDER BY interval
```

This SQL produces the query shown in Example E-6.

Example: E-6 Blank interval investigation: check the QAPYJWSTS file

Current

Interval number	TDE status	Current wait	wait duration
146	A	320	2,912,207
147	I	320	12,852,389
148	A	320	172,562
149	A	320	4,703,268
150	A	320	9,617,863
151	A	320	2,851,697
152	A	320	176,951
153	A	320	6,557,608
154	A	320	2,784,058
155	A	320	113,292
156	A	102	3,184,794
157	A	102	4,635,844
158	I	102	14,662,648
159	I	102	24,663,345
160	A	49	6,905
177	A	103	73,891
178	I	103	10,103,931

Analysis of this query reveals:

Interval 147

The task count is in idle status (I) as it used 0 CPU in this interval. The current wait type (ENUM) is 320 which is 'common TIMI object checker: seize object' from the QIDRWCH/QAIDRJWENM file. This reveals that this is some sort of seize-related wait. Checking the wait bucket file QSYS/QAPYJWBKT for ENUM 320, we see that this ENUM is associated with wait bucket 15, which is the Seize contention wait bucket.

The cumulative time spent in this single on-going seize wait is 12,852,389 microseconds.

The interval size in this collection is 10 seconds.

The in progress wait shown at the end of interval 146 in the QAPYJWSTS file is 2,912,253 microseconds, which reveals that interval 147 was slightly less than 10 seconds.

There is no way to determine the exact duration of interval 147, as the only place that the start and end timestamps are stored is in the QAPYJWTD file and there is no record in that file for this interval and task count.

Interval 158

The task count is in idle status (I) as it used 0 CPU in this interval. The current wait type (ENUM) is 102, which is a shared seize' (from the QIDRWCH/QAIDRJWENM file). Again, this reveals that this is some sort of seize-related wait. Checking the wait bucket file QSYS/QAPYJWBKT for ENUM 102, we see that this ENUM is associated with wait bucket 15, which is the Seize contention wait bucket.

The cumulative time spent in this single on-going seize wait is 14,662,648 microseconds.

The interval size in this collection is 10 seconds.

The in-progress wait shown at the end of interval 157 in the QAPYJWSTS file is 4,635,844 microseconds.

Interval 159

The task count is in idle status (I) as it used 0 CPU in this interval. The current wait type (ENUM) is 102, which is a shared seize (from the QIDRWCH/QAIDRJWENM file). Once again this reveals that this is some sort of seize-related wait. Checking the wait bucket file QSYS/QAPYJWBKT for ENUM 102, we see that this ENUM is

associated with wait bucket 15, which is the Seize contention wait bucket.

The cumulative time spent in this single on-going seize wait is now 24,663,345 microseconds, as this individual wait has now spanned two 10-second intervals.

The in-progress wait shown at the end of interval 157 in the QAPYJWSTS file is 4,635,844 microseconds. If you deduct this from the cumulative wait the answer is approximately 20 seconds or two full intervals of wait time (no interval is precisely 10 seconds).

You can now determine what happened in interval 178, as it is similar to interval 147, but with a different low-level seize wait in progress.

Job run/wait profile: how to fill in blank intervals

Restriction: The functionality described in this section became publicly available during 1Q 2005 in an update to iDoctor for iSeries Job Watcher, available at:

https://www-912.ibm.com/i_dir/idoctor.nsf/

This topic is based on an early driver of this new-in-2005 function and assumes you have installed this update on your iSeries system.

The Job Watcher developers are providing the capability to fill in any blank intervals on the Job run/wait time signature graphs and to correctly represent the worst types of waits on the Collection overview time signature graph.

This capability is evoked through the GUI as shown in Figure E-5 on page 272, by selecting the **Create idle threads graph files** option.

Tip: If you want to compare the original collection values with the possible gaps to an adjusted one produced by Create idle thread graphs files function, first copy your collection to a new library where it is the only Job Watcher collection before you run this function.

No original data is lost or adjusted. This is because the Create idle thread graphs files function creates new files and uses them instead of the originally collected files:

- ▶ QAIDRJWIG2
- ▶ QAIDRJWIS0
- ▶ QAIDRJWIS2

In our example, we copied the collection to a new, empty library called FILLEDWAIT.

We did this because after the Create idle thread graphs files function completes, the new QAIDxxxx files are automatically identified by Job Watcher and used for graphing purposes. New file QAIDRJWIG2 is used by the GUI for graphing job run/wait signatures instead of QAPYJWTDE, so you cannot compare the “before-running-this” and “after-running-this” versions of the same graph unless you first have copied the collection to another library.

Another of the created files, QAIDRJWIS2, is used by the GUI for interval summary graphs. This file not only fills the wait gaps but also speeds up the graphing process as all the detail records no longer have to be queried.

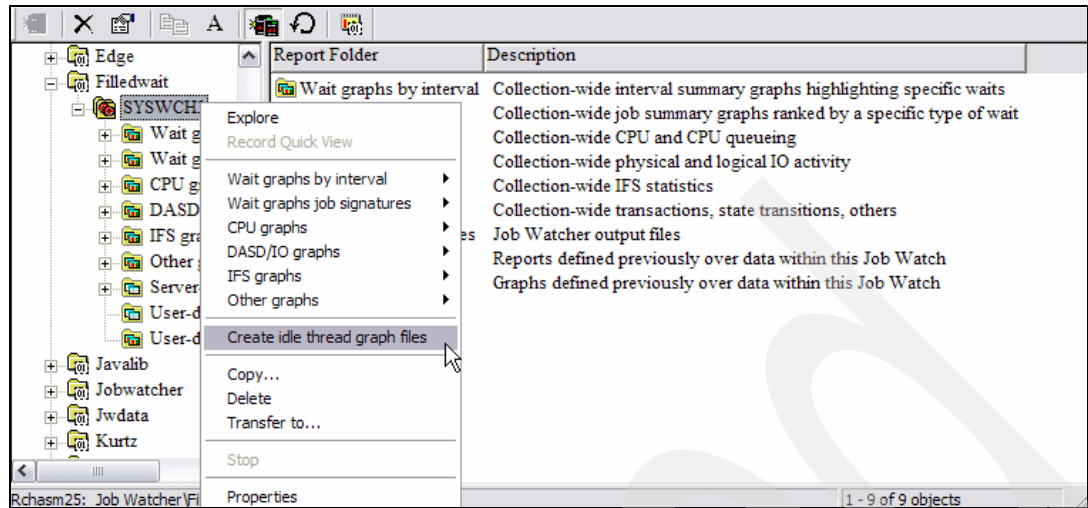


Figure E-5 Running the interval filler

This option submits job name QCRTWCHSUM to batch that runs the CRTWCHSUM command.

When the create graph files function has completed, you receive the confirmation message shown in Figure E-6.

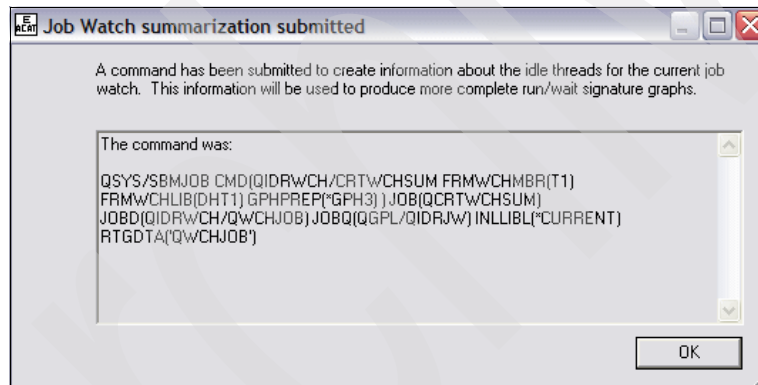


Figure E-6 Interval Filler confirmation message

When the CRTWCHSUM command has completed, it has created the following new files in your collection library:

- ▶ QAIDRJWIG2 - CRTWCHSUM Interval graph record file
- ▶ QAIDRJWIS0 - CRTWCHSUM Control file
- ▶ QAIDRJWIS2 - CRTWCHSUM Interval summary file

The Job Watcher graphing support for intervalized wait signatures, such as the Collection overview time signature and the Job run/wait time signature, will *always* use these new files instead of the QAPYJWTDE file whenever the collection member exists in these files. This is why we copy the collection to a new, empty library before running the CRTWCHSUM command, so that we can compare the before and after graphs.

Before and after graphs for filled blank interval support

We compare the Collection overview time signature graphs in Figure E-7.

- ▶ The graph at the top of the figure shows the collection-wide wait profile for our data when the CRTWCHSUM command has not yet been run.
- ▶ The graph at the bottom of the figure shows the collection-wide wait profile for the same data after the CRTWCHSUM command has been run.

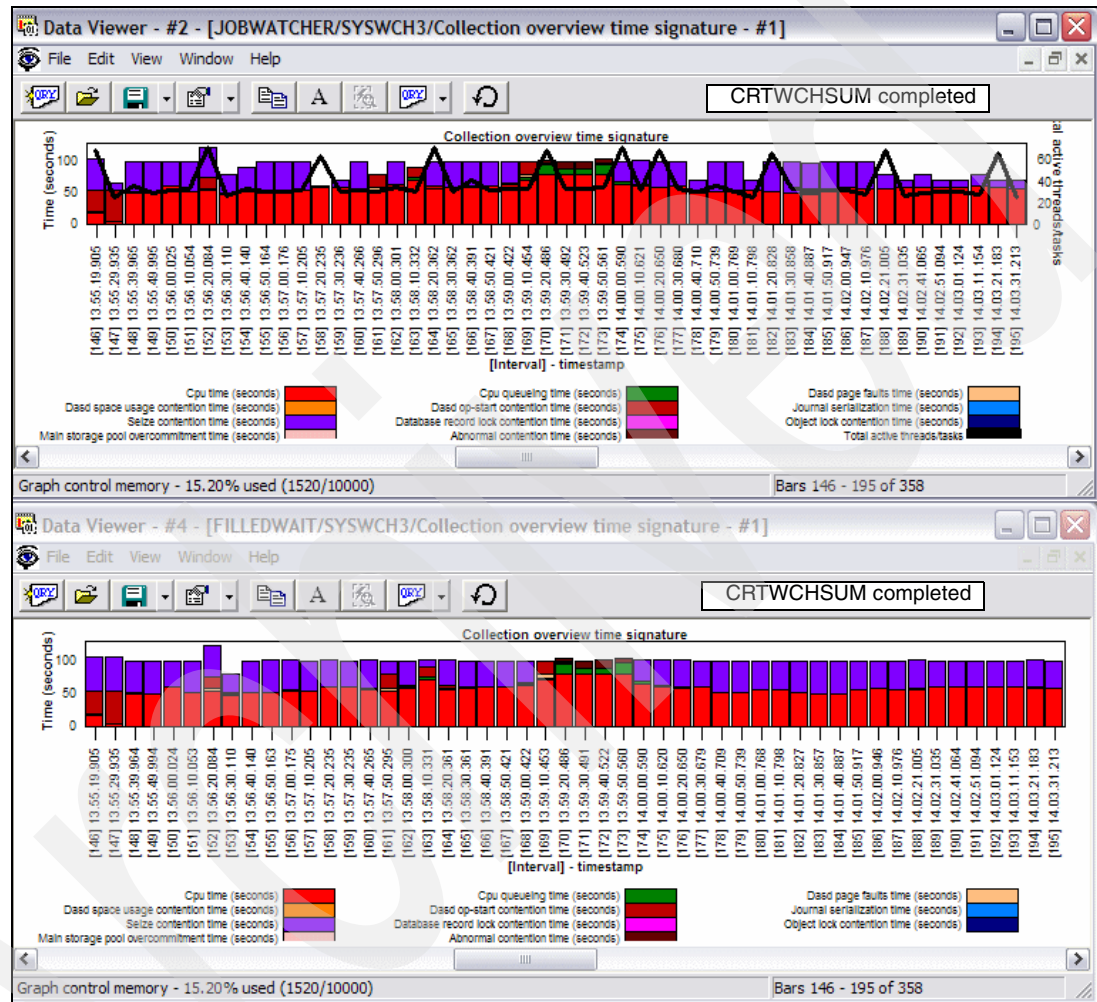


Figure E-7 Filled blank waits comparison: Collection overview time signature

Notice that the seize wait time in intervals such as 147, 158, 159, and 161 is now correctly represented in the lower graph.

We next compare the Job run/wait time signature graphs in Figure E-8. The top graph shows the intervalized wait data before the CRTWCHSUM command is run. The bottom graph shows the intervalized wait data after the CRTWCHSUM command is run.

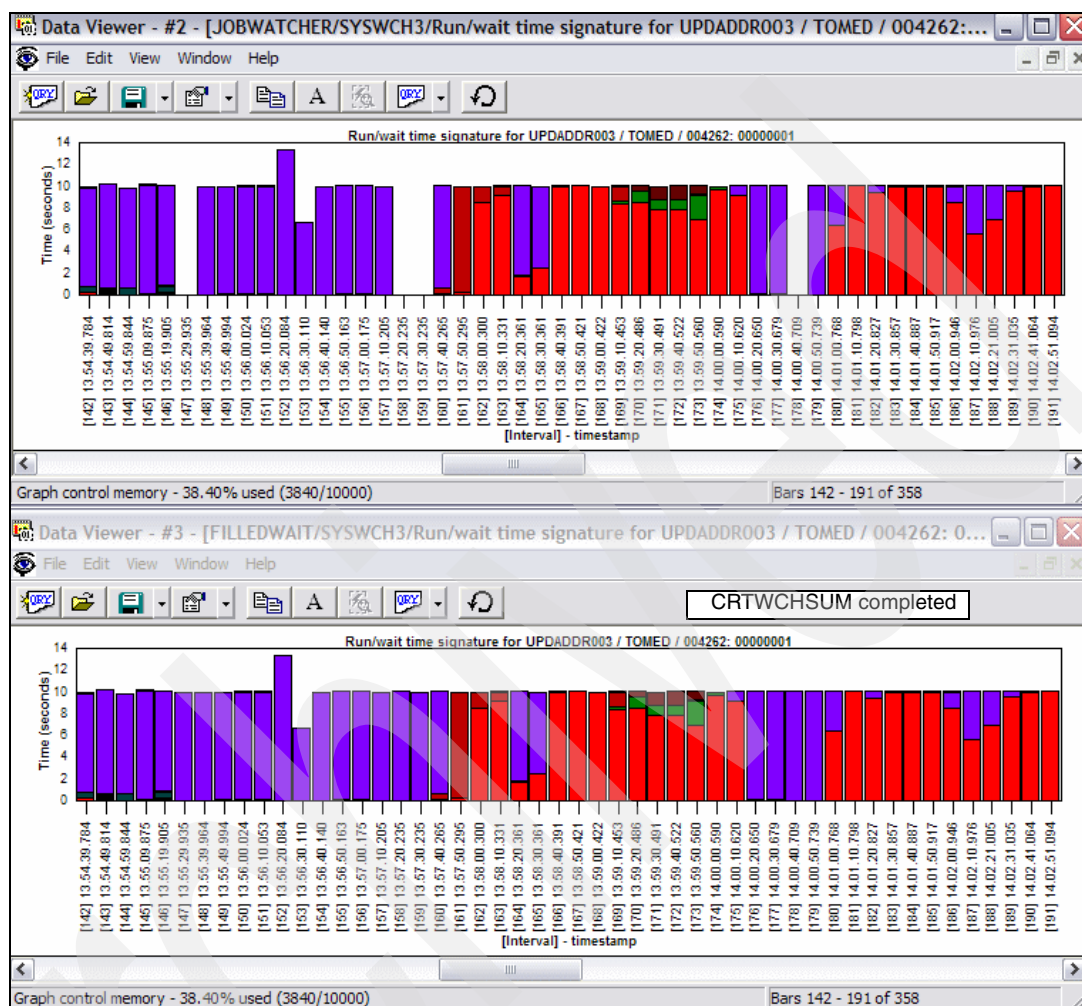


Figure E-8 Filled blank waits comparison: Job run/wait time signature

Notice that the seize wait time in intervals such as 147, 158, 159, and 178 is now correctly represented in the lower graph.

Job Watcher limitations

Job Watcher is a sampling tool, not a tracing tool, so there are limitations to what it can see at the time of writing this publication. The Job Watcher developers are aware of these limitations and will be addressing them in future releases.

How many jobs on the system?

Tools such as Collection Services “see” all jobs on the system, irrespective of whether those jobs last 2 hours or 2 seconds. Job Watcher, however, only records data for a job if the job existed for at least two consecutive snapshots during a Job Watcher collection.

Consider Figure E-9 on page 275.

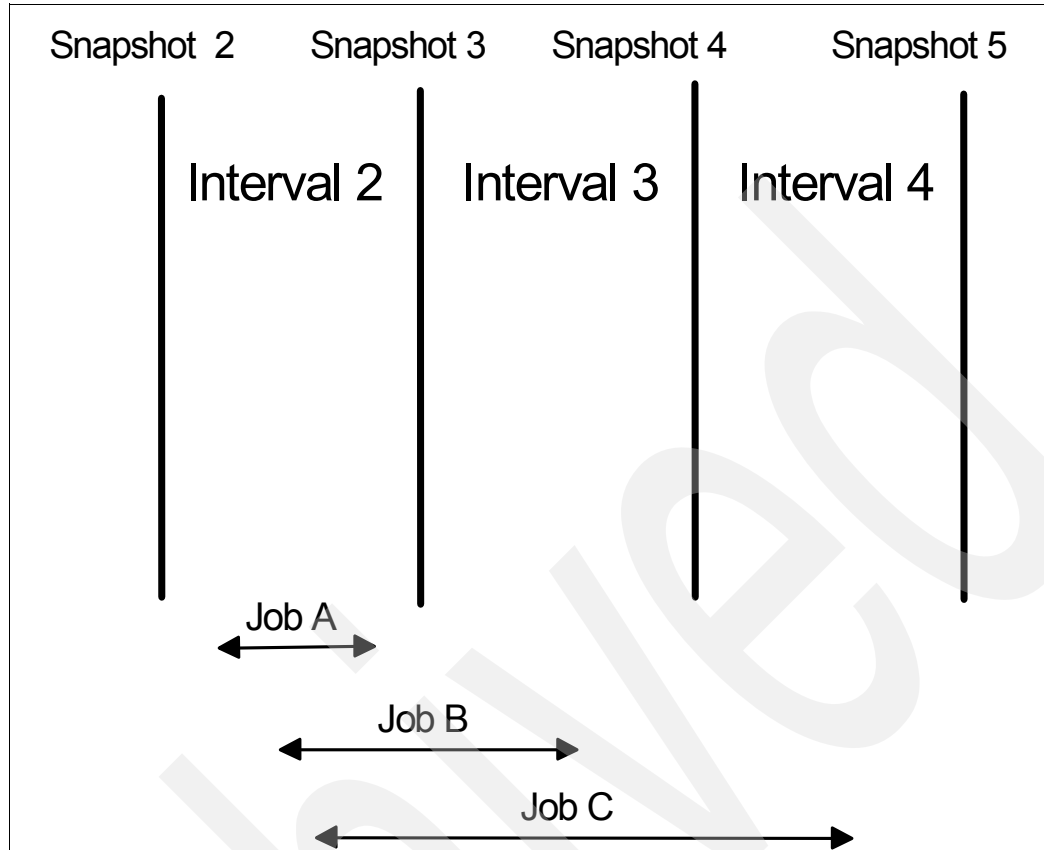


Figure E-9 Jobs that Job Watcher may not see

For ease of discussion assume that jobs A, B, and C are all single threaded.

- **Job A:** This job starts and ends within a single interval and therefore is not known to the Job Watcher collector at snapshot time. There will be no Job Watcher delta statistics (CPU, DASD Operations, Waits, etc.) for Job A in either the QAPYJWTDE or QAPYJWSTS files. There will be a count of the Job A type instances in the QAIDRJWIS2 file, field INTJBIANT - Interval Job Initiations and Terminations. It is a count, but Job Watcher does not know the resources used when the **Create idle thread graph files** option is taken as described for Table E-1 on page 259, Figure E-5 on page 272, Figure E-6 on page 272.

Record type "I" will have a value in field INTJBIANT of file QAIDRJWIS2, representing initiations and terminations counted, but not contributing to CPU, DASD, Wait, and so forth, within interval statistics.

- **Job B:** This job starts in interval 2 and ends in interval 3. There are no delta statistics for this job as it did not exist for at least two intervals. There will be one record in the QAPYJWSTS file for this job for interval 2 and no records for this job in the QAPYJWTDE file.
- **Job C:** This job starts in interval 2 and ends in interval 4. There are delta statistics in QAPYJWTDE for this job only for interval 3. There will be one record in the QAPYJWSTS file for this job with a TDESTSTATUS field value of T for interval 4.

The Job Watcher statistics for jobs A, B, and C will never match those collected by Collection Services, performance trace tools such as Performance Explorer (PEX), or printed and displayed by the Performance Tools for iSeries licensed program. Remember, Collection Services and PEX record all jobs started, running, or stopped while they are running.

We have seen Job Watcher data from customer systems that have many jobs that start and end within a very short time period. Knowing about this limitation could dictate the collection interval duration that you would choose to use (short duration intervals would capture more short-lived jobs), as well as whether to run Collection Services along with the Job Watcher collection.

From a Job Watcher view, you can detect that many more jobs and threads have been started and completed than shown in its graphical reports or your own queries, by querying the Job Watcher file QAPYJWSTS file produced by Job Watcher's collection.

You can check the maximum task count number per interval. Although job numbers are not assigned in numerical sequence, task count numbers are.

Run the query in Figure E-7 against the QAPYJWSTS file to see:

- ▶ The highest task count number in use by interval
- ▶ The total number of unique task counts in use during that interval

Example: E-7 Finding spikes in job creation activity

```
SELECT interval,
max(taskcount) as maxtaskcount, count(distinct(taskcount))
as different_taskcounts
FROM qapyjwsts
GROUP BY interval
ORDER BY 2 desc
```

This query produces results similar to those shown in Example E-8. You should consider running Collection Services and producing reports on the associated performance database files to corroborate the Job Watcher output.

Example: E-8 Job creation or termination activity results

Interval number	MAXTASKCOUNT	DIFFERENT_TASKCOUNTS
1,176	517,250	12,578
1,175	517,219	12,574
1,174	517,196	12,570
1,173	517,171	12,569
1,172	517,140	12,566
1,171	517,118	12,569
records removed for brevity		
445	500,803	10,068
444	500,789	10,063
443	500,759	10,048
442	500,744	10,048
441	500,705	10,027
440	500,663	9,999
439	500,608	9,977
438	500,582	9,982
records removed for brevity		
9	488,434	13,920
8	488,272	13,903
7	488,118	13,893
6	487,971	13,891
5	487,829	13,892
4	487,670	13,887
3	487,516	13,883

2	487,192	13,864
1	486,751	13,854

This collection was for 2 hours with 10-second intervals.

Notice that the maximum task count has increased by 30,000+ between intervals 1 and 1176.

This indicates that there is a very large amount of job create/delete activity (start/end) on this system.

Archived

Glossary

Dispatched Time. The term used to represent both the time a thread or task has been dispatched to run on a processor but not yet actually using the processor (ready to run but waiting for a processor) plus the time the thread or task actually was running in the processor. There is no actual metric that measures the separate time of “dispatched to a processor but waiting to use it.” The book discusses other Job Watcher metrics or graphics that indicate that threads or tasks have been waiting for an available processor, just not the length of time.

Thread. A separate and unique flow of control within a job. A thread runs a procedure asynchronously to other threads running within the job. The term thread is an above-TIMI term; the LIC actually uses the term task count.

A single job can have one or many threads. It always has one single primary thread (also called an initial thread) and may have zero to many secondary threads.

The work performed by a job is the sum of work performed by all threads within that job plus the work that is done for that job by other system tasks or jobs.

A LIC task does not have threads.

Task count. A unique numeric identifier assigned by the microcode to every “dispatchable unit of work.” Every job-thread and LIC task has a unique task count value, assigned in ascending sequence, starting when the system or partition is IPLed (started or activated). A task count value is unique within a partition while that partition is active.

A task count is associated with a TDE (see following text), which is a control block representing an individual task or individual thread within a job. Within a job, there is one task count that represents the primary thread and zero to many unique, different task counts that represent each of any secondary threads.

Note that at the microcode level each task count has a run priority that was either established at the operating system level (such as via the job’s run priority parameter) or according to microcode implementation requirements.

TDE. (Task Dispatcher Element) An LIC control block that anchors every dispatchable unit (thread or task) to be run in the system. The tasking component of the iSeries system uses the TDE when dispatching work (assigning work to run on a processor. Note that many of the performance metrics are scoped to a thread or task in a TDE and harvested by Job Watcher from the TDE.

In the context of this book, a TDE is associated with a single thread of a job or a single LIC task. The TDE contains both the task count and the accounting data necessary for running the thread or task in priority among other threads/tasks and for basic performance analysis.

Snapshot. A Job Watcher collection sample of the selected job-threads and tasks.

Interval. The delta time between snapshots. An interval number is stored in Job Watcher files for chronological sequencing. The interval size is based on the data collection interval selected.

Eye Catcher. A four-character (three meaningful characters followed by a blank character) short description acronym indicating an individual type of low-level wait status of a job, thread, or task within the system. It may or may not indicate why the job is not progressing. For example, an eye catcher of Rex represents a wait for a request for an exclusive seize and has an ENUM of 101.

ENUM. How the system internally identifies an individual type of low-level wait.

A single ENUM has only one eye catcher; however, a single eye catcher may be associated with several different ENUMs.

An ENUM is also known as a block identifier (where block is used as in “blocking the path”).

Wait bucket. A bucket number is also known as a bucket identifier, wait bucket, block bucket, or queueing bucket identifier.

There are 32 separate bucket identifiers for every job/thread and task on the system.

Each bucket identifier has two parts, counts, and durations.

At Version 5 Release 3 there are 199 different ENUMs, each of which represents an individual low-level wait. These 199 ENUMs are mapped into the 32 buckets, hence the counts and durations of several related ENUMs are mapped into the same bucket. This grouping into buckets is done to minimize overhead of extra information being stored in the TDE.

Archived

Related publications

The documentation listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 282. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Maximum Performance with WebSphere Application Server V5.1 on iSeries*, SG24-6383
- ▶ *Striving for Optimal Journal Performance on DB2 Universal Database for iSeries*, SG24-6286

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ Primary iDoctor for iSeries Web sites

Main: http://www.ibm.com/eserver/iseries/support/i_dir/idoctor.nsf

Downloads: https://www-912.ibm.com/i_dir/idoctor.nsf/downloadsV5R3.html

- ▶ IBM @server iSeries Information Center Web site

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm>

At this Web site, select the iSeries i5/OS release level and language. On the initial page you can expand the left navigation area. Search for the topic you are interested in and expand it. Examples include:

- Performance tools

In the main window, search for **performance** to find information about the Performance Tools for iSeries Licensed Program.

- Programming

Expand **Java**. One of the elective topics is **Java platform**. Select one of the articles.

- SQL Performance

The easiest way to find the PDF *iSeriesDB2 Universal Database for iSeries Database Performance and Query Optimization*, is to select the center link for **Printable PDFs and manuals** and then search on **performance**.

- ▶ iSeries Performance Management Web site

<http://www.ibm.com/eserver/iseries/perfmgmt>

Select **Resource Library**. Look under the various headings, such as Performance Papers and Articles. You can select one of several different links, such as one labelled Tips and Tuning WebSphere and Java applications.

- ▶ Job Watcher trial version

https://www-912.ibm.com/i_dir/idoctor.nsf/JWTrialAgreement.html

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Using the SQL examples in this redbook

We have purposely shown most of our example SQL statements in this book in Example type figures. If you have Adobe Reader to read the PDF, you can easily copy and paste the SQL example to some text editing tool, such as Microsoft® Word, or to an actual SQL interface, such as Job Watcher's Data Viewer - SQL Query view interface.

For the PDF of this redbook, go to:

<http://www.redbooks.ibm.com/>

Search for SG24-6474 for a link to the PDF.

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

*MGTCOL object 14

Numerics

2757 or 2780 disk controller 154
5250 commands 241
5722-PT1 9

A

ABNORMAL CONTENTION (bucket description) 205
access code 56
Access Job Watcher 55
access path 133
access paths 147
activation group 118
 statistics 19
 usage 118
activation group information 204
activation groups 61
Active Job Watcher 77
active JVM 186
active SQL cursors 127
active SQL statement
 viewing 129
active SQL statements and host variables tip 60
Add Connection option 54
ADDPDACS command 250
Advisor 15
AIDRJWIG2 file 271
ASP threshold 52
asynchronous database reads 211
automatic scaling 84
automatically submit a Job Watch 251
auto-refresh reports 84
auxiliary storage threshold 78

B

B2B activity monitor 9
bad wait discussion 26
bars per page 84
batch job trace report 15
BIRTHDAY 210
blank interval 89, 267
 in Job Watcher data 271
block bucket 25, 201
block identifier. 201
bucket 1 (Time Dispatched on a CPU) 114
bucket identifier 25, 201
bucket number 25
 (wait bucket) definition 201

C

call stack 3, 19, 58–59, 169–170
 for jobs with active SQL statements 141
 Object create profile data 185
 reported by task count 200
call stack level information 141
Call stack tab 99
cast as decimal 137
CAST as DECIMAL (SQL) 153
CAST as operator 235
CFGPFRCOL 14
class handle 187
Collect as fast as possible 257
 Job Watcher option 161
collect data for idle jobs/threads at first interval option 258
collection engine 52
collection jobs, tasks options 52
collection name 57
collection object 9
Collection overview time signature graph 85, 127
Collection Services 8, 34
column headings 235
commitment control rollbacks 217
communication lines 16
communications data 19
component report 15
components of performance 2
configure Job Watcher startup 161
connecting to your system 54
Copy option 81
copy-LIC wait buckets 260
correct Job Watcher collection member name 139
CPU
 available 267
CPU “misusage” 160
CPU components 3
CPU intensive application and journaling 157
CPU time 33, 114
CPU time definition 114
CPU utilization 16
CPYJWCOL command 81, 247
create idle thread graph files 275
create idle thread graphs files 271
creating a total SQL statements work file 138
CRTPFRTA command 14
CRTWCHSUM command 272, 274
current activation group count 218
current or last low level wait type 212
current state 212
current user profile 69
CURRSTATE 212

D

- DASD Page Faults
 - Faults Graph by Job graph 146
- DASD/IO graphs 114
- Data 108
- data area operations count 217
- data collection 58
- data collection options 58, 60–61
- data collection options - SQL 60
- data queue receives 217
- data queue sends 217
- Data Viewer 108, 134
 - Query definition interface 134
 - running 135
 - SQL Query view prompt 139
 - SQL view (full function SQL) 134
 - use fully qualified file/table 139
 - x and y axes scaling 156
- DB Monitor 11, 133, 140
- DB2 Access Paths 36
- DB2 Multi-tasking 34
- DBL3 tasks 3
- DEBUG (RTVSTKDTA command) 251
- Definition (parameter settings) 57
- delete option 81
- delta thread count 216
- deltacpu (dispatched CPU time) 154
- derived field (SQL) 138
- disk 16
- disk busy percentage 114
- dispatched for use of CPU 148
- dispatched time 33, 279
- display I/O transactions 217
- Display Performance Data (DSPPFRTA) command 14
- Display Performance Data command 15
- DLTJWCOL command 81, 247
- Domino server 25
- DROP TABLE 134
- DSPPFRTA (Display Performance Data) command 14
- Dump Java Virtual Machine (DMPJVM) command 172

E

- Enable automatic scaling (iDoctor preference) 84
- End Performance Collection (ENDPFRCOL) 9
- Ending Options window 74
- ENDPFRTA
 - End Performance Trace 10
- enhancements to Job Watcher at V5R3M0 52
- ENUM 25, 30, 201
- ENUM definition 201
- ENUM mappings 204
- ENUM to queueing bucket mappings 205
- exclusive seize 201
- eye catcher 25, 201
- eye catcher definition 36–47

F

- field headings 235
- field selection

- Toggle Selected 226
- File monitor 9
- fonts 84
- FTPJWCOL command 81, 248
- fully opened SQL cursors 218
- future enhancements 255

G

- garbage collection 172
- garbage collector 190
- gates 5
- getting started 51
- GO PERFORM 14
- Graph History function 8–9
- graphs
 - user-defined 238
- graphs and reports
 - example 1 85
 - example 2 94
- graphs to view 238
- Group By tab 229
- GUI 52
- GUI component of Job Watcher 78

H

- Hardware multi-threading (HMT) 22
- hardware multi-threading (HMT) 33
- Heap Analyzer 5, 12–13, 172
- heap growth - Java 172
- heap information for an activation group 204
- Heap Watch - Java 173
- high availability journal performance feature #42 (journal caching) 144
- High Availability Journal Performance option 154
- holding job-thread/task type 213
- Holding thread/task tab 103
- host variables 127, 129
- How Many Jobs on the System? 274
- how to obtain Job Watcher 53
- HTTP server 25

I

- Idle threads graph files 81
- iDoctor
 - client workstation 78
 - documentation 58
 - GUI 78
 - preferences 82
- iDoctor for iSeries Heap Analysis Tools for Java 12
- iDoctor for iSeries Job Watcher 12
- iDoctor for iSeries PEX Analyzer 12
- IFS 100, 207
- IFS directory creates 218
- IFS directory read count 217
- IFS open count 218
- Include all jobs and tasks option 63
- index 133
- interval 25, 201

- requested versus actual 214
- Interval Details window 90
- Interval duration (startup option) 57
- invocation stack 19
- IOPs 17
- iSeries Navigator Performance Tools Plug-In 15
- iSeries Navigator SQL Performance Monitor 133

J

- Java resources for more information 159
- Java threads analysis 169
- Java Virtual Machine (JVM) 174
- job creation or termination activity results 276
- job function 216
- Job monitor 9
- job names
 - capturing for all jobs 256
- Job report 15
- Job signature ranked by CPU graphics tip 111
- Job signatures ranked by seize 95
- Job signatures ranked by seizes graphics tip 111
- Job state transitions tab 105
- Job summary graphs/reports options 84
- Job Watch Name option 57
- Job Watcher 12
 - files created by 199
 - GUI component 78
 - Job/Task Options 63
 - jobs 12
 - libraries 53
 - limitations 274
 - obtaining 53
 - Update history Web page 255
 - V5R3 enhancements 52
- Job Watcher and Java analysis 160
- Job Watcher client GUI 78
- Job Watcher collected data 12
- Job Watcher collection
 - starting with GUI 54
 - starting with Wizard 56
 - stopping 78
- Job Watcher Collection Process details 257
- Job Watcher Collection Wizard 56
- Job Watcher data
 - querying 222
- Job Watcher Data Collection Options for Java 162
- Job Watcher download 20
- Job Watcher GUI 78
- Job Watcher link 53
- Job Watcher start 56
- Job Watcher subsystem 76
- Job Watcher tip 20
- Job Watcher view 79
- Job Watcher Wizard Summary 76
- jobs/threads/tasks ranking 74
- JOBTYP 216
- Journal Bundle Owner wait (JBo) 154
- Journal Bundle wait (JBw) 154
- journal caching 144–145, 154
- journal serialization 129

- journal serialization wait 148
 - zero time value? 153
- journal wait 148, 153
- JVM 176

L

- L2 cache 22
- L3 cache 22
- last executed SQL statement option 127
- launch button 56
- LDIO close count 217
- LDIO commit count 217
- LDIO open count 217
- LDIO rollback count 217
- LDIOCOMIT 218
- LDIOOTHR 218
- LDIOOTHR (other logical disk I/O (non read/write)) 216
- LDIORD 218
- LDIORD (logical disk I/O Read) 216
- LDIOROLLB 218
- LDIOs (logical database operations) 107
- LDIOWRT 218
- LDIOWRT (logical disk I/O Write) 216
- LDTAAP 217
- LDTAQRCV 217
- LDTAQRCV value 218
- LDTAQSND 217
- LDTAQSND value 218
- left bracket symbol (indicates an array object) 179
- LEFT OUTER JOIN 138
- Library (Job Watcher startup option) 57
- LIC assist tasks 3
- Licensed Internal Code (LIC) 23, 71
- Lock report 15
- locks 5
- Logical Disk I/O (LDIO) 216
- logical file pages 147
- logical I/O 107
- Logical I/Os tab 107
- LPAR shared/partial processors 34

M

- main collection database files 258
- main storage 16
- Management Central 8
- Maximum data collected (stopping Job Watcher) 78
- Maximum data to collect (Job Watcher option) 58, 161
- Maximum intervals reached (stopping Job Watcher) 78
- maximum task count number per interval 276
- memory leak 172
- Message monitor 9
- MODTYPE - Module name and VRM 251
- MODTYPE (RTVSTKDTA command) 251
- monitors (performance) 9

N

- Naming convention changes 52
- No Procedure Found message 120

nonintrusive 99, 120

O

object class - Java 179
Object Class Name column (Heap Watch) 179
Object counts/sizes by class loader 186
Object create profile 13, 173, 179, 186
Object handle 187
Object locks 17
Object root finder 173–174, 186
Object table dump 13, 173–174, 186
Object waited on tab 101
optimizer 11
Other waits (bucket description) 205
OVRDBF command 204

P

page faults 36, 129
page faults - Java 172
Page Faults graph 116
pCreateStdActivation 121
percentage (SQL calculation) 153
performance component
 summary 6
performance database files 9
Performance Explorer 10, 16
Performance Management for iSeries 15
Performance Management Web site 159
performance monitors 9
Performance Tools for iSeries license program product
 (5722PT1) 14
Performance Tools for iSeries Licensed Program Product
 (5722PT1) 2, 9
Performance Tools Plug-In 15
Performance Tools reports 15
performance trace data 10, 15
Performance Trace Data Visualizer 12
PEX 10, 16
PEX Analyzer 12–13
PEX Task Switch Trace 266
PEX Trace 11, 13, 16
Physical I/Os tab 106
PM eServer iSeries 14
PM/400 14
pool ID 73
Pool report 15
POWER5 22
POWER5 technology 34
Primary iDoctor for iSeries website 281
Primary Y-axis tab 239
priority change flag 210
procedures 118
processor sharing 35
program activation group information 204
programs 118
PRTPEXRPT 13
PSA entries in use 214
PTDV 12

Q

QAIDRJWCPU file 202
QAIDRJWDFN file 202
QAIDRJWIS0 file 271
QAIDRJWIS2 file 271
QAIDRJWRI file 202
QAIDRJWSTK file 141, 170
QAPYJW file name prefix 53
QAPYJWAIGP (activation group) file 121
QAPYJWAIGP file 204
QAPYJWAIPA file 204
QAPYJWBKT file 202
QAPYJWINTI file 202
QAPYJWJVM file 202
QAPYJWJVTH file 202
QAPYJWPRC file 202, 216
QAPYJWPROC file 203
QAPYJWRUNI file 202
QAPYJWSKJB file 204
QAPYJWSQL file 129, 135–136, 203
QAPYJWSQLO file 203
QAPYJWSQLP file 203
QAPYJWSTK file 141
QAPYJWSTS file 202
 advanced analysis 269
QAPYJWTDE file 135–136, 153, 202, 267
 advanced analysis 268
 and blank intervals 268
 cumulative statistics 208
 delta statistics 208
 fields 208
QAPYJWxxx file names (replacing QPYRTxxx) 53
QAYPETDE file 144
QAYPYJWAIHP file 204
QAYPYJWSKTC file 204
QAYPYJWSQLH file 203
QAYPYJWSTK file 203
qcount13 (journal serialization wait bucket) 153
QCRTWCHSUM job 272
QDBGET 216
QDBGETDR 216
QDBGETKY 216
QDBGETM 216
QDBGETMQO 216
QDBGETSQ 216
QDBGETSQO 216
QDBPUT 216
QDBPUTDR 216
QDBPUTM 216
QDBPUTMX 216
QIDRGUI library (replaces QYPBAS) 53
QIDRJW 76
QIDRJW subsystem 78
QIDRJWENM file 204
QIDRWCH library (replaces QPYRTJW) 53
QPFRAJ system value 105
QPRCMLTTSK system value 35
QPYRTJW library 53
QPYSWJOB 78
qtime01 (dispatched to CPU time) 154

- Query definition interface 134, 222
- Query Definition window 236
- query definitions, working with 233
- querying Job Watcher data 222
- queueing bucket identifier 25, 201
- queueing bucket identifier mapping 204
- queueing primitives 32
- Quick view tab 90, 97
- QWCHJOB 78
- QYPBASE library 53
- QYPSPFCOL 14
- QZDASOINIT jobs 69

R

- RCOUNT - Interval or QREC (or *ALL) (RTVSTKDATA command) 251
- real-time performance tools 6
- record selection 228
- REDBOOKDH library 140
- Redbooks Web site 282
 - Contact us x
- reports and graphs
 - example 1 85
 - example 2 94
- Resource report 15
- Retrieve Call Stack Data command 141, 170, 250
- Retrieve Stack Data command 119
- Reuse these settings option 227
- root finder analysis 186
- root finder collection 187
- root finder time limit 188
- RPG 118
- RSTJWCOL command 249
- RTVSTKDATA command 119, 141, 170, 250
- Run/wait signature for a job graph 267

S

- sampling - Job Watcher 99
- sampling based performance tool - Job Watcher 29
- sampling tool
 - Job Watcher 120
- saving your new query definition 232
- SAVJWCOL command 249
- SBMJOB 119
- secondary Y-axis 240
- Seize contention graph 87
- seize time in microseconds 211
- seize wait time 273
- seizes 5
- Select fields option 79
- Select specific jobs and tasks option 63
- SELECT statement 134, 235
- selecting specific jobs and tasks 64
- Series Navigator SQL Performance Monitor 140
- service programs 118
- shared seize 201
- Simultaneous Multi-threading (SMT) 22
- simultaneous multi-threading (SMT) 34
- snapshot 24, 201, 258

- sockets communication data collection option 62
- Sort By tab 228
- specific jobs and tasks, Job Watcher option 64, 163
- SQL 60, 102
 - analysis example 127
 - example of ENUMs 204
 - numeric data truncation 153
 - performance 126
 - performance monitors 11
 - SQL statement query 137
- SQL CREATE table 134
- SQL Query view 134, 222, 233
- SQL Query view interface (Data Viewer) 134
- SQL statement in progress 214
- SQL statements
 - active 127
 - and host variables collected by Job Watcher 19
 - most popular 139
 - most popular by task count 140
- SQL Visual Explain 11
- SSTAR technology 33
- Start Performance Collection (STRPFRCOL) 9, 14
- Start Performance Tools (STRPFRT) 14
- Start Service Tools (SST) 172
- STKMBRNAME (RTVSTKDATA parameter) 251
- stop option 78
- STRDBMON 11
- STRJRN JRNCACHE parameter 145
- STRPFRCOL command 14
- STRPFRT 14
- STRPFTRC
 - Start Performance Trace 10
- Subsystem (Job Watcher option) 72
- Summarize (Job Summary option) 84
- SWIN task status 263
- Switched Available for Dispatch (SWAFD) 267
- Switched In (SWIN) 266
- Switched Out Queue (SWOQ) 266
- SWOQ task switch out status 263
- Synchronous 211
- synchronous database reads 210
- synchronous database writes 211
- synchronous non-database reads 211
- System monitor 9
- System report 15
- system-level tools 6

T

- task count 71, 90, 148, 170, 200
 - 24, 200
- task count number 276
- task count state changes 266
- task dispatchable unit priorities 218
- Task Dispatcher Element
 - TDE 24, 200
- task ID 200
- task name 68
- TCOUNT (RTVSTKDATA command) 251
- TDE 201
- TDE definition 24

- TDE status 215
- TDETYPE T (thread) 209
- tdeusecs (interval time) 153
- Technology Independent Machine Interface (TIMI) 3, 23, 200
- THRDSTATUS 209–210
- thread ID 209
- thread status 209–210
- Threads using CPU this interval 167
- time dispatched 33
- Time Dispatched on a CPU (bucket 1) 114
- time interval bars (Job Watcher graphs) 129
- time limit reached 78
- TIMI
 - See Technology Independent Machine Interface
- total threads count for the job 216
- total time in current wait in microseconds 213
- totcountnn, tottimenn fields in QAPYJWTDE file 154
- TOTSTMTCNT file/table 138
- Transaction (Interval details) 98
- Transaction report 15
- True CPU Microseconds (useconds) Used 35
- two gears icon 78

U

- Update history Web page 255
- user space operations or index operations count 217
- user-defined graphs 238
- Using the SQL examples in this redbook 282

V

- viewing Job Watcher data 78
- virtual processor 34
- Visual Explain 11

W

- wait analysis 27
- wait bucket 25, 29, 201
 - definitions for Job Watcher 30
 - disclaimer 32
 - journal serialization waits 153
- wait bucket 1 - Dispatched time 33
- wait bucket 10 - DASD (other reads or writes) 38
- wait bucket 11 - DASD operation start contention 39
- wait bucket 12 - Mutex/semaphore contention 40
- wait bucket 13 - Journal serialization 40
- wait bucket 14 - Machine level gate serialization 40
- wait bucket 15 - Seize contention 41
- wait bucket 16 - Database record lock contention 41
- wait bucket 17 - Object lock contention 42
- wait bucket 18 - Other waits 43
- wait bucket 19 - Main storage pool overcommitment 43
- Wait bucket 2
 - CPU queuing 35
- wait bucket 2 - CPU queuing 35
- wait bucket 20 - Java user (including locks) 44
- wait bucket 21 - Java JVM 44
- wait bucket 22 - Java (other) 45

- wait bucket 23 - Socket accepts 45
- wait bucket 24 - Socket transmits 45
- wait bucket 25 - Socket receives 45
- wait bucket 27 - IFS pipe 46
- wait bucket 27 - Socket (other) 46
- wait bucket 28 - IFS (other) 46
- wait bucket 29 - Data queue receives 46
- Wait bucket 3 - Total block time 36
- wait bucket 30 - MI queue (other) 46
- wait bucket 31 - MI wait on events 47
- wait bucket 32 - Abnormal contention 47
- Wait bucket 4 - Reserved 36
- Wait bucket 5 - DASD (page faults) 36
- Wait Bucket 6 - DASD (non-fault reads) 37
- Wait bucket 7 - DASD space usage contention 37
- Wait bucket 8 - Idle/waiting for work 37
- Wait bucket 9 - DASD writes 38
- wait bucket definitions 30
- Wait buckets tab 104
- wait component computation 26
- Wait graphs by job 110
- wait object name 213
- wait points 32
- waits at LIC microcode level 29
- waits for asynchronous writes 211
- waits that span multiple intervals 267
- watch name 78
- WCHJOB command 199, 202, 242
- WCHLIB - Watch library name 250
- WCHLIB (RTVSTKDTA command) 250
- WCHMBR - Watch member name 250
- WCHMBR (RTVSTKDTA command) 250
- wizard
 - Job Watcher collection wizard 56
- work file TOTSTMTCNTcleanup 140
- Work with Active Jobs (WRKACTJOB) 2
- Work with Disk Status (WRKDSKSTS) 2
- Work with System Activity (WRKSYSACT) 17
- Work with System Status (WRKSYSSTS) 172
- WRKACTJOB command 7, 12, 17, 160
- WRKDSKSTS command 7
- WRKSYSACT command 7, 12, 17, 19, 34, 160
- WRKSYSSTS command 7

X

- XPF (i5/OS) 210



IBM iDoctor for iSeries Job Watcher: Advanced Performance Tool

(0.5" spine)
0.475" <-> 0.873"
250 <-> 459 pages



IBM iDoctor for iSeries Job Watcher: Advanced Performance Tool



**A tough tool for
tough performance
problems**

**Use it when other
performance tools
do not identify the
problem**

**A how-to book for the
“average” iSeries
performance expert!**

This IBM Redbook is intended for those familiar with most of the IBM-provided iSeries performance tools that are generally available through the i5/OS operating systems commands and iSeries Navigator interfaces and the additional-cost Performance Tools for iSeries, 5722-PT1, licensed program.

iDoctor for iSeries is a set of software performance analysis tools and associated services that extend your ability to evaluate the health of your iSeries-based system by gathering detailed information and providing automated, graphical analysis of this data. One of these tools, Job Watcher, is the key next-step advanced tool for analyzing detailed performance data.

This book:

- Gives an overview of Job Watcher and most other IBM-provided iSeries performance measurement and management tools.
- Describes the components of performance and how Job Watcher provides access to detailed performance data.
- Provides examples of Job Watcher functions and its GUI in three applications: traditional RPG, SQL, and Java.
- Provides Job Watcher collected data file and field definitions, and SQL query examples of this data beyond Job Watcher's array of graphical reports and drill-down information.

This book's objective is to enhance the performance analyst's proficiency in using Job Watcher as a key tool in the performance analysis tool kit.

**INTERNATIONAL
TECHNICAL
SUPPORT
ORGANIZATION**

**BUILDING TECHNICAL
INFORMATION BASED ON
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:
ibm.com/redbooks**

SG24-6474-00

ISBN 073849237X