# Optimizing System z Batch Applications by Exploiting Parallelism

Managing your batch window

Optimizing batch technology

Studying a hands-on case

Martin Packer
Dean Harrison
Karen Wilkins

Redpaper

ibm.com/redbooks

International Technical Support Organization

# Optimizing System z Batch Applications by Exploiting Parallelism

August 2014

REDP-5068-00

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

**First Edition (August 2014)**

This edition applies to Version 2, Release 1, Modification 0 of z/OS (product number 5650-ZOS).

This document was created or updated on August 20, 2014.

# Contents

**Notices** . . . vii
Trademarks . . . viii

**Preface** . . . ix
Authors . . . ix
Now you can become a published author, too! . . . x
Comments welcome . . . x
Stay connected to IBM Redbooks . . . x

**Chapter 1. Introduction** . . . 1
1.1 Background . . . 2
1.2 What batch job splitting is . . . 2
1.3 Why batch job splitting is important . . . 2
1.4 When to re-engineer your jobs . . . 4
1.5 New applications . . . 4
1.6 Analysis . . . 5
1.6.1 Job analysis . . . 5
1.6.2 Environment analysis . . . 6
1.7 Issues . . . 6
1.8 Other forms of batch parallelism . . . 6
1.9 The approach that is used in this publication . . . 7

**Chapter 2. Data management considerations** . . . 9
2.1 What is new in DB2 11 . . . 10
2.2 Commit processing . . . 10
2.3 Parallel processing . . . 10
2.3.1 DB2 parallelism . . . 11
2.3.2 Parallelism for partitioned and nonpartitioned table spaces . . . 11
2.3.3 Enabling parallelism . . . 11
2.3.4 DB2 11 improvements . . . 12
2.4 Data skew . . . 12
2.5 Buffer pools . . . 13

**Chapter 3. Application design considerations** . . . 15
3.1 Overall approach . . . 16
3.1.1 A model for job splitting . . . 16
3.2 Fan-out . . . 16
3.2.1 DFSORT OUTFIL record selection capabilities . . . 17
3.2.2 Checking data . . . 19
3.3 Replicated job instances . . . 20
3.4 Fan-in . . . 20
3.5 Reporting . . . 21
3.5.1 Report data collation . . . 21
3.5.2 Report formatting . . . 22

**Chapter 4. Considerations for job streams** . . . 25
4.1 Scenarios . . . 26
4.2 Degrees of parallelism . . . 26
4.3 Data marshalling . . . 26

4.3.1 Unnecessary fan-in and subsequent fan-out jobs . . . . . . . . . . . . . . . . . . . . . . . . 27
4.4 Sequences of jobs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.5 Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**Chapter 5. Operational considerations** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.1 Preparing for parallel batch processing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.1.1 Key components of a parallel batch structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.1.2 Naming considerations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.2 JCL splitting techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2.1 Base JCL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.2.2 Simple JCL splitting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.2.3 Minimizing JCL maintenance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.2.4 Control data communication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.2.5 Removing individual job members . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.2.6 Merging output data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.3 Scheduling techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.3.1 Static versus dynamic scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.3.2 Static parallel scheduling techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.3.3 Dynamic parallel scheduling techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.3.4 Initialization contention . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.3.5 Minimizing scheduling maintenance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.3.6 Reducing operational impact . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.4 Recovery and rerun . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.5 Workload service assurance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.5.1 Milestone alerting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.5.2 Critical path monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

**Chapter 6. Resource management considerations** . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
6.1 Processors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.2 Memory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.2.1 Concurrent sorting operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.3 Disk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.4 Tape . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.5 Initiators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.5.1 WLM managed initiators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

**Chapter 7. Case study** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.1 Case study scenario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
7.1.1 Application synopsis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
7.1.2 Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
7.1.3 Instrumentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
7.2 Test iterations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
7.2.1 Base case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
7.2.2 First refinement: DB2 COMMIT processing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
7.2.3 Second refinement: Partitioning the updated DB2 table . . . . . . . . . . . . . . . . . . . 93
7.2.4 Third refinement: Increasing the number of partitions . . . . . . . . . . . . . . . . . . . . . 94
7.2.5 Fourth refinement: Aligning partitions to input . . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.2.6 Fifth refinement: In-memory table space . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
7.2.7 Sixth refinement: In-memory table space with first reference . . . . . . . . . . . . . . . 97
7.2.8 Seventh refinement: In-memory table space and increased buffer pool size . . . . 98
7.3 Relationship to customer installations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

**Appendix A. Application modernization opportunities** . . . . . . . . . . . . . . . . . . . . . . . 101
Repurposing reporting data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

Creating an XML file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
Creating a JSON file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
Modernizing report generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

**Appendix B. Operational supplement** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
Obtaining sample exits and the SOE toolkit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
REXX utility JCLCATDD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

**Related publications** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
IBM Redbooks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
Other publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
Online resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
Help from IBM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116

# Notices

## Trademarks

# Preface

This IBM® Redpaper™ publication shows you how to speed up batch jobs by splitting them into near-identical instances (sometimes referred to as *clones*). It is a practical guide, which is based on the authors' testing experiences with a batch job that is similar to those jobs that are found in customer applications. This guide documents the issues that the team encountered and how the issues were resolved. The final tuned implementation produced better results than the initial traditional implementation.

Because job splitting often requires application code changes, this guide includes a description of some aspects of application modernization you might consider if you must modify your application.

The authors mirror the intended audience for this paper because they are specialists in IBM DB2®, IBM Tivoli® Workload Scheduler for z/OS®, and z/OS batch performance.

## Authors

This paper was produced by a team of specialists from the United Kingdom (UK) working at the International Technical Support Organization, Poughkeepsie Center.

**Martin Packer** is a Consulting IT Specialist working for the IBM Worldwide Banking Center of Excellence. He has 28 years of mainframe performance experience, having consulted with dozens of customers in that time. He has contributed to many IBM Redbooks® publications over the past 23 years, and has presented at many conferences. He built a number of tools for processing mainframe performance instrumentation, most notably in the areas of batch window reduction, IBM Parallel Sysplex® Performance, and IBM DB2 Tuning. He holds a Bachelor's degree in Mathematics and Physics and a Master's degree in Information Technology, both from University College London.

**Dean Harrison** is a Workload Automation Specialist working worldwide for the IBM Software Group, with a UK base. He has 28 years of experience in the batch management field, having worked with nearly all the flavors and versions of the Tivoli Workload Automation product family on many platforms. He has written and presented extensively on the Tivoli Workload Scheduler for z/OS Program Interface. He is also the lead developer of the Scheduling Operational Environment (SOE), a PIF-based toolkit that is now in use by over 100 customers worldwide.

**Karen Wilkins** is a Client Technical Professional specializing in the DB2 Tools and DB2 Engine on IBM System z® in the UK. She has 26 years of experience in IT, starting out as a junior programmer, then becoming an independent consultant specializing in application development. She managed a development and support team for a large car manufacturer in the UK. She has worked in various industries, including insurance, manufacturing, and the retail sector, and various application migration projects. She has worked at IBM for three years as a Senior IT specialist, supporting DB2 on System z customers in the UK. She is actively involved in supporting DB2 user groups in the UK and has presented at a number of their meetings.

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

Bob Haimowitz
**International Technical Support Organization, Poughkeepsie Center**

Mark Wisniewski
**IBM Poughkeepsie**

## Now you can become a published author, too!

## Comments welcome

## Stay connected to IBM Redbooks

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  `https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm`

- Stay current on recent Redbooks publications with RSS Feeds:

  `http://www.redbooks.ibm.com/rss.html`

1

# Introduction

This chapter introduces the topic of *batch job splitting*. It explains why batch job elapsed time is an issue for some enterprises. It covers the relationship between processor technology changes, workload growth, and the challenge of meeting service level agreements (SLAs). Subsequent chapters describe the technologies that are presented in this chapter and present the results of a series of tests that were performed.

## 1.1 Background

Processing power has increased enormously throughout the history of computing. Although the improvements in processor speed, memory access times, and input/output (I/O) throughput have occurred at different times, they all historically have shown vast improvements. However, recent industry trends show a reduced growth rate in processing speed while workload growth does not show any sign of diminishing. At the same time, batch windows are getting shorter; today's batch window is much shorter than it used to be.

Taken together, the batch window challenge and the continued growth in business workload volumes mean that batch processing must run faster.

Reducing batch elapsed times in an environment where processing speed growth is reducing requires a new approach. For many installations, increased batch *parallelism* is required. For some workloads, techniques similar to those that are referenced in 1.8, "Other forms of batch parallelism" on page 6 suffice. These other forms of parallelism generally are easier to implement, and less likely to break applications. For others, splitting selected jobs is necessary to support that growth.

In most cases, only a small subset of an installation's jobs (typically large jobs on the critical path) must be tuned to use more parallelism. The job-splitting approach relies more on overall system *throughput* capacity than on single thread speed.

## 1.2 What batch job splitting is

Consider a single-step batch job, which processes a complete set of data using a single program. We call this job the *original job*.

In many cases, it is feasible to run multiple jobs, each running the same program, against the data. The objective is to process the data faster by dividing the work among all these jobs and running them in parallel. These jobs are sometimes called *clones*. This publication refers to them as *instances*.

Marshalling data, and indeed the jobs themselves, requires care. If you are marshalling data (gathering and transforming data into a computer format), additional jobs are required. In some cases, the data is partitioned, with each instance running against a subset of the data. In other cases, the data is shared, with each instance accessing the entire set of data. In practice, most job-splitting scenarios employ a mixture of partitioned data access and shared data access.

## 1.3 Why batch job splitting is important

This section describes computing industry trends in processor design. The processor is a good example of a resource whose rate of performance increasefalls short of many installations' workload growth requirements.

For many years, much of the growth in processor speed came from a reduction in the clock cycle time (or increase in clock frequency). For example, the processors of the IBM zEnterprise® 196 (z196) have a 5.2 GHz clock frequency, which is an 18% increase in speed compared to the IBM System z10® Enterprise Class (4.4 GHz). However, the processors of the IBM zEnterprise EC12 (zEC12 with 5.5 GHz) saw only a moderate 6% increase over those of the IBM zEnterprise 196.

Nonetheless, the typical (observed) single processor speed increase between z196 and zEC12 was substantially more than the change in clock cycle time indicates. The difference largely is accounted for by design changes rather than clock speed. Acceleration through design, rather than raw electronic speed, is typical of how speeds are increased between processor generations.

**Note:** In the previous paragraph, the term “design” incorporates two themes:

- Increased processor cache sizes
- More intelligent processor logic

Processor cache sizes are governed by semiconductor feature sizes: the smaller the feature, the more cache can be incorporated in the same space. Clock speed similarly is affected by feature size. While feature sizes are decreasing, the rate of decrease is slowing. Cache size in the future likely is to grow more slowly.

Also, the overall system capacity (for a maximally configured system) increased between z196 and zEC12 by more than the single-processor increase. This was caused by two factors:

- A client can order a zEC12 with a maximum of 101 configurable processors, and a z196 can be ordered with a maximum of 80 configurable processors.
- Design changes between processor generations often produce better multiprocessor ratios, meaning that additional processors provide relatively more capacity.

These two factors are interrelated. Better multiprocessor ratios allow the effective use of more processors.

Overall processor capacity largely is irrelevant to single batch jobs, but it *is* important for overall batch workloads. But many individual batch jobs depend on thread speed.[1]

A number of processor architectures support *multithreading*. With multithreading, a single processor can appear to software as though it were more than one processor. Typical processor designs support 2-way, 4-way, or even 8-way multithreading.

Industry figures suggest a processor with 2-way multithreading provides approximately 1.4 times the throughput of a single-threaded processor. A 4-way multithreading processor provides approximately 2.0 times the throughput of a single-threaded processor.

Although multithreading increases the throughput from a single processor, it provides less thread speed. For example, 1.4 times the throughput with two threads suggests that each thread is 0.7 times the speed of the single-threaded case.

The industry trend is towards performance gains being achieved through increased parallelism, rather than single thread speed increases.

[1] *Thread speed* is clock speed times the throughput gain from threading, divided by the threads per processor. For more information about this topic, go to the following website: http://www.ibmsystemsmag.com/mainframe/administrator/performance/relative_capacity/

## 1.4 When to re-engineer your jobs

There are three major scenarios where you might consider job splitting:

- When long-term planning suggests it might be necessary.
- In response to a short-term batch performance crisis.
- When modernizing the application.

Although the first two scenarios are performance-driven, the third scenario is centered around improved function or reliability.

Long-term planning involves deciding the likely infrastructure design for the next few years and designing your batch environment to take advantage of it. For example, your design might be that you plan to use vastly more abundant memory over the next few years.

Short-term crisis resolution means fixing imminent or already present performance problems. In principle, regular batch run monitoring avoids such problems. In reality, installations sometimes experience them.

The administrator of an installation might decide to renovate an application to extend its life, add a function, or address modern requirements, such as supporting mobile devices. One such requirement might be replacing line-mode printer reports with Portable Document Format (PDF) reports. Another might be allowing other applications to use the raw reporting data with the IBM Worklight® server. For more information about this function, see the IBM information center found at the following website:

http://www-01.ibm.com/support/knowledgecenter/api/redirect/wrklight/v5r0m5/index.jsp

## 1.5 New applications

Although this publication addresses existing batch applications, similar considerations and design techniques apply equally to new batch applications. In particular, design new applications to support high levels of parallelism. Designing applications without inhibitors to multi-threading allows them to scale optimally. It allows the administrator of an installation to decide about the degrees of parallelization to run, giving the administrator more technology choices and operational flexibility. The cheapest and easiest time to remove multi-threading inhibitors is when the application is still in the design phase.

This publication describes traditional techniques for multi-threading. New application development styles can also be used to promote multi-threading. For more information, see the following resources:

- *A More Effective Comparison Starts With Relative Capacity*, found at:

  http://www.ibmsystemsmag.com/mainframe/administrator/performance/relative_capacity

- *How Simultaneous Multithreading Could Affect System z*, found at:

  http://www.ibmsystemsmag.com/mainframe/trends/IBM-Research/smt_mainframe/

## 1.6 Analysis

Installations typically run many thousands of jobs at night. Splitting a job is an expensive and fragile task. So, it is important to reduce the list of jobs to split to a manageable number. To produce a small candidate list, carefully analyze your jobs. It is also important to understand the environment in which the jobs run. This section describes both job selection and the processing environment.

### 1.6.1 Job analysis

Chapter 15, "Approaches and techniques to reduce the batch window", in *Batch Modernization on z/OS*, SG24-7779, describes how to discern which batch jobs' performance is important to your batch window. Doing a critical path analysis establishes which jobs are on the critical path from the start of the window to its finish. Any reduction in the run time of a job on the critical path shortens the window. So, establishing which jobs are on the critical path is important.

A PC-based productivity tool to help with this analysis is IBM System z Batch Network (zBNA). It provides a means of estimating the elapsed time for batch jobs that are based solely on the differences in processor speeds for a base processor and a target processor, the number of engines on each system, and system capacities (data sharing is not considered). zBNA provides a powerful graphic demonstration of the z/OS batch window, as shown at the following website:

http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/PRS5132

There are several questions to ask about a batch job:

1. Does it need to run faster?
2. Are there less intrusive techniques for speeding it up than splitting it up?
3. Would less intrusive techniques produce the long-term elapsed time reduction the job needs?
4. How many instances does the job need to be split into?
5. Is it feasible to split the job?

The answer to question 1 is "no" for most jobs. This leads to a high degree of filtering of jobs, with only a short list needing tuning.

Although the answer to question 2 might be "yes", the answer to question 3 might be "no"; consider the long term. Less intrusive techniques can deliver limited benefits.

For example, a tuning exercise might reduce a job's run time by 30%. This might be enough to sustain the next two years' growth. But the technique usually cannot be repeated. Therefore, it will not support the next five years' growth. For jobs where other techniques suffice, splitting is unnecessary. So, questions 2 and 3 further reduce the list of jobs that need the splitting approach.

Question 4 is relevant only for the jobs that you decide to split. Even if the answer to question 4 today is "two", you should design your splitting scheme so that four, eight, 16, or even 32 instances can be accommodated. This paper describes how to do this task optimally.

Question 5 is the most difficult one to answer; it requires even more careful analysis than question 2 does. This paper provides guidance about this topic. While writing this paper, we encountered cases where splitting a job was not feasible.

### 1.6.2 Environment analysis

*Batch Modernization on z/OS*, SG24-7779 describes how to assess your batch environment. The methodology described in it (which was originally described in *System/390 MVS Parallel Sysplex Batch Performance*, SG24-2557) recommends analyzing your environment before considering individual applications and jobs.

Analyze your environment to understand if you have adequate resources for splitting to be successful. For example, a z/OS system that has no spare memory is unlikely to support more parallelism without an increase in paging. Similarly, I/O bandwidth might limit the scope for parallelism.

Although your current environment might not have enough resources to support batch job splitting, consider whether your system design should change to support splitting. An example is to ensure that a new system is memory-rich where the current one is not.

## 1.7 Issues

This publication is primarily a description about how to resolve issues when splitting a batch job and, in particular, a single step. Issues can arise in several areas:

- Data management is described in Chapter 2, “Data management considerations” on page 9.
- Application design is described in Chapter 3, “Application design considerations” on page 15.
- Operations are described in Chapter 5, “Operational considerations” on page 29.
- Resource management is described in Chapter 6, “Resource management considerations” on page 71.

The topic of performance is described in all of these chapters. The ability to effectively and efficiently split a job depends on resolving these types of issues.

As an example of effectiveness, a job should run much faster when expanded from two instances to four. Another effectiveness consideration is the economic cost. For example, a serious increase in processor usage is likely unacceptable.

As an example of efficiency, the process of expanding from two instances to four should be quick, error-free, and result in maintainable code.

## 1.8 Other forms of batch parallelism

z/OS and its subsystems, such as DB2, have a rich variety of capabilities that support batch parallelism. Many of these capabilities are mentioned in *Batch Modernization on z/OS*, SG24-7779 and *System/390 MVS Parallel Sysplex Batch Performance*, SG24-2557.

Although this publication focuses on single-step cloning, there are many other approaches. Those other approaches might be preferable because they are less intrusive. Here are examples of those other approaches:

- DB2 Query Parallelism
- Breaking up a multi-step job into several jobs, each with fewer steps

- Breaking inter-step dependencies that are caused by sequential data sets, using BatchPipes/IBM MVS™
- Writing multiple output data sets in parallel with DFSORT OUTFIL
- Processing sequential data sets with parallel I/O streams, using DFSMS Sequential Data Striping

None of these techniques are as intrusive as job splitting, which generally requires application program changes. Even DB2 Query Parallelism and DFSORT OUTFIL are easier to implement than splitting.

All these techniques require significant analysis.

## 1.9 The approach that is used in this publication

This publication contains two types of information:

- General techniques to make job splitting successful
- A case study where some of the techniques are deployed

In our example, we wrote a batch job whose characteristics are similar to many installations' batch jobs. We then split it repeatedly, resolving any issues that we encountered at each stage. In the process, we used standard instrumentation to guide our efforts. This case study is presented in Chapter 7, "Case study" on page 75.

**Note:** Some techniques that we used are specific to the application, so we do not describe them in depth in this paper. For example, the basis on which you partition data is application-specific.

2

# Data management considerations

This chapter looks at some of the data management considerations when you split application logic to run across multiple instances. However, it does not cover physical or logical database design or managing DB2 performance across a subsystem or member.

## 2.1 What is new in DB2 11

The premier IBM data management subsystem on System z is IBM DB2 for z/OS. A common theme in recent DB2 releases is the aim of reducing the total cost of ownership (TCO). Major contributors to that goal have been efforts to accomplish the following tasks:

- Improve operational efficiency by delivering processor savings.
- Provide unsurpassed resiliency through increased scalability and improved availability.
- Improve performance by reducing CPU processing time and optimizing query processing without causing significant administration or application changes.

For a complete guide to what is introduced in DB2 11, see *DB2 11 for z/OS What's New,* GC19-4068, or the online IBM Knowledge Center at the following website:

http://www-01.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/com.ibm.db2z11.doc.wnew/src/wnew/db2z_wnew.dita

## 2.2 Commit processing

Commit processing involves taking database checkpoints to which you can return if you run into problems or roadblocks. If your application is not written with DB2 **COMMIT** and optionally **ROLLBACK** logic incorporated, then one of the first problems you might encounter is a DB2 deadlock or timeout as you try to run multiple parallel instances. As you add commit logic, you must establish a balance between too many commit points, which can increase the processor usage of the program, and enough commit points so that shared resources can be accessed concurrently by different logical units of work.

**ROLLBACK** processing can be processor-expensive and time-consuming. The work that must be backed out also must be reapplied when the process is restarted.

There are other considerations with adding **COMMIT/ROLLBACK** processing to program logic. For example, if flat files are being read or written, then you probably must include additional logic to support file repositioning if there is a restarted job that failed previously. This can complicate the JCL that is needed to run the program. You also might need to have slightly different logic paths for when a program starts for the first time or when it restarts.

## 2.3 Parallel processing

This publication describes running multiple instances in parallel. This is not the same as DB2 parallelism, which is described briefly in 2.3.1, "DB2 parallelism" on page 11.

Partitioning tables can enable DB2 to take more advantage of parallel processing. However, how much benefit partitioning can provide depends on whether the queries that you are trying to improve are I/O- or processor-intensive. This is why it is important to make sure that you take adequate measurements of performance data and analyze this data to highlight where there is scope for improvement.

If you decide that partitioning a table is likely to improve your ability to run multiple instances, you must establish how many partitions are optimum. If this number must change because of data growth or the need to increase parallelism, the number of partitions can be modified with the **ALTER TABLESPACE** statement. For information about how to determine the number of partitions that are required, based on whether the queries are I/O- or processor-intensive, see *DB2 11 for z/OS Managing Performance,* SC19-4060.

The size of the partitions also contributes to how well DB2 can use parallelism. DB2 uses the size of the largest physical partition to determine the potential degree of parallelism. Therefore, by making the partitions all a similar size, DB2 can create a potentially higher degree of parallelism than if there were one oversized partition in relation to the others.

### 2.3.1 DB2 parallelism

DB2 can initiate multiple parallel operations when it accesses data from a table or index in a partitioned table space.

Query I/O parallelism manages concurrent I/O requests for a single query, fetching pages into the buffer pool in parallel. This processing can improve significantly the performance of I/O-bound queries. I/O parallelism is used only when one of the other parallelism modes cannot be used.

Query CP parallelism enables true multitasking within a query. A large query can be broken into multiple smaller queries. These smaller queries run simultaneously on multiple processors accessing data in parallel, which reduces the elapsed time for a query.

### 2.3.2 Parallelism for partitioned and nonpartitioned table spaces

Parallel operations usually involve at least one table in a partitioned table space. Scans of large partitioned table spaces have the greatest performance improvements, where both I/O and central processor (CP) operations can be carried out in parallel.

Each of these types (non-partitioned, partitioned, range-partitioned, and partition-by-growth) of table spaces can take advantage of query parallelism. Parallelism is enabled to include non-clustering indexes. Thus, table access can be run in parallel when the application is bound with **DEGREE (ANY)** and the table is accessed through a non-clustering index.

If you choose to partition to improve performance, there are some general considerations that must be taken into account, which are documented in *DB2 11 for z/OS Managing Performance,* SC19-4060.

However, DB2 might not always choose parallelism, even if you partition the data.

### 2.3.3 Enabling parallelism

Even if you do all of the preparation work to enable DB2 to choose parallelism, DB2 cannot do so unless you are running with two or more tightly coupled processors, and certain **BIND** or **REBIND** options and special registers are in effect. Furthermore, there are many restrictions that prevent parallelism from being used.

### 2.3.4 DB2 11 improvements

DB2 11 delivers the following improvements:

- Performance improvements are made by reducing processor usage when table spaces that have many partitions are accessed.

  In earlier versions of DB2 for z/OS, you might experience performance degradation after increasing the number of partitions in a table space.

- Availability is improved by enabling the limit keys for a partitioned table space to be altered without an outage. When you change the limit key values with an online alter limit key, the data remains available and applications can continue to access the data.

  In earlier versions of DB2, when you changed the limit key values, all affected partitions were placed in REORG-pending (REORP) status, which means that the data was unavailable until you ran the REORG utility.

  In DB2 11, in most circumstances, changing the limit key values has become a pending definition change. This means that the limit key changes are not materialized until you run a REORG. Until then, affected partitions are placed in an advisory REORG-pending (AREOR) status.

  There are restrictions on this behavior. It is available in the following situations:

  - Range-partitioned universal table spaces
  - Table spaces that are partitioned (non-universal) with table-controlled partitioning

## 2.4 Data skew

One of the issues we avoided in our testing for this publication was the issue of *data skew*. We were working with data that might not reflect a real-life scenario. In our testing, each of our instances ran in a fairly uniform manner. However, when working with data that can be skewed (for example, sales data that is partitioned by month), it might contain increased volumes at certain times of year.

An implication of data skew is that performance can be negatively affected because of contention on I/O and other resources. You might also find yourself with some partitions being close to their maximum size limits while others have excess space available. You might be able to redistribute the data by adjusting the partition boundaries to reduce the skew, or you can let DB2 to choose the new boundaries by using the **REBALANCE** keyword on a **REORG TABLESPACE**.

Managing the data in DB2 is not the only way to handle such data skew. If instances consistently take different processor and elapsed times to run, you might want to use scheduling options to allow a long running instance to always run in parallel with potentially more than one other instance in a leg, as shown in Figure 2-1 on page 13.

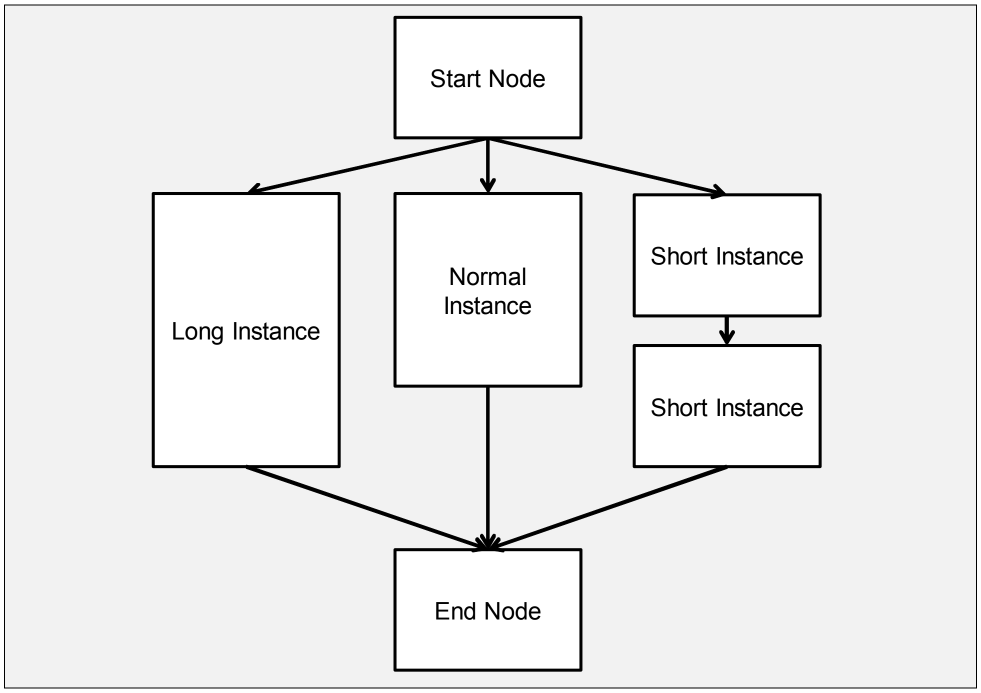


*Figure 2-1 Diagram showing one option to handle different data volumes in instances*

## 2.5 Buffer pools

As a general rule of thumb, large buffer pools can improve performance. DB2 is efficient in the way it searches buffer pools. This can be a good option if you are constrained by I/O resources.

A feature that was introduced in DB2 10 is the option to fix certain tables and indexes in memory by specifying **`PGSTEAL(NONE)`** on your buffer pools. Upon first reference, the pages are loaded into the buffer pool and remain there. It is important to make sure that you have enough real memory available if you want to use this option. Close monitoring is advised if you switch on this option. If you see significant paging activity, either increase the amount of real storage or decrease the size of the buffer pools. If insufficient real storage exists to back the buffer pool storage, the resulting paging activity might cause performance degradation.

**Note:** For more information about DB2 and buffer pools, there are many publications available, including the following ones:

- *DB2 11 for z/OS Technical Overview*, SG24-8180
- *DB2 10 for z/OS Performance Topics*, SG24-7942

3

# Application design considerations

This chapter describes considerations in application design when splitting a single-step batch job. However, it does not describe application development methodology.

## 3.1 Overall approach

Examine in detail how data flows through a batch job before designing a split version. Making data continue to flow properly upon splitting is a major challenge.

A typical single-step batch job is not a single step. Generally, it contains the following steps:

- Steps to set up for the main processing step. These steps might perform actions such as the following ones:
  - Deleting data sets from previous runs.
  - Allocating fresh versions of these data sets to which the main processing step will write.
  - Verifying the integrity of the input data.
  - Sorting input data.
- The main processing step, which typically does the following actions:
  - Reads and updates data.
  - Produces a report.
- Steps to clean up after the main processing step.

For these reasons, in this publication the term *single real step* is used to denote a job with only one significant processing step. This publication mainly addresses single real step jobs, while acknowledging that in real applications there are often jobs with multiple real processing steps. Breaking a multi-step job into single steps makes splitting, as this paper defines it, much easier. The techniques that are described here are also applicable to multi-step jobs.

### 3.1.1 A model for job splitting

This publication presents an example model of how to split a job. It contains a number of elements:

- A fan-out stage
- Replicated job instances that perform the bulk of the processing
- A fan-in stage
- A reporting stage

Some of these elements might not be present in a specific case. For example, there might not need to be a fan-in stage or a reporting stage if there is nothing for them to do. These stages do not necessarily correspond to jobs. For example, the fan-in stage and the reporting stage might be in the same job or even the same step. The remainder of this chapter describes these stages.

## 3.2 Fan-out

If the original job processes input data, the data must be split. If there is input data, a separate job to split the data must run before the instances run. In this publication, this job is called the *fan-out* job. There might not be a fan-out job if there is no data that must be split.

An alternative is to alter the instances to read all the input data but select only those records of interest. For small input data sets that are read only once, this might be feasible. For larger or more intensively read data sets, I/O contention might become an issue. Also, this approach requires extra logic in the instance program.

In many cases, the input data is in more than one data set and each data set has a different logical organization. In these cases, each data set must be split in an appropriate manner. For simplicity, this section assumes a single data set. Treat the multiple data set case the same as the single data set case, that is, once for each data set.

How to split depends on a number of factors. This section presents some of the considerations.

Balance is not just about splitting input data into even subsets. Even subsets might lead to uneven run times. You must analyze the effect on run times when you select a method to split your data. In addition, try to split data in a way that is logical and maintainable.

Splitting a large data set can take significant time and thus delay processing. The splitting criteria periodically must be reworked. For fast processing and easy reworking of splits, try to use `DFSORT COPY` or `SORT` processing:

- If the sequence of records is important but the data is not already sequenced, use `SORT`. The sequence might be required by application logic or might be beneficial for performance.
  - If the sequence is required by application logic, the sort already should be present. Add `OUTFIL` statements to this sort with the appropriate selection criteria to split the output.
  - If the sequence is suggested for performance reasons, it might not be present. For example, analysis of the original job might suggest a sort can help trigger DB2 prefetch I/O. Use the split as the opportunity to perform this sort.
- If the sequence of records is unimportant or the records are already correctly sequenced, use `COPY`, avoiding the more expensive `SORT`. Use `OUTFIL` statements to split the data set.

## 3.2.1 DFSORT OUTFIL record selection capabilities

It is beyond the scope of this publication to describe all the record selection capabilities of `DFSORT OUTFIL`. For information about these capabilities, see *z/OS DFSORT Application Programming Guide,* SC23-6878. This publication shows some examples of things you can do with `OUTFIL`.

### Explicit record selection

Select records according to whether specific fields meet certain criteria by using statements, such as the one shown in Example 3-1.

*Example 3-1 Explicit record selection*

```
OPTION COPY
OUTFIL FNAMES=HIPHOP,INCLUDE=(Honorific,EQ,C'Dr',AND,Surname,EQ,C'Dre')
END
```

In Example 3-1 on page 17, the output DD name is `HIPHOP` and the two symbols are defined in the `SYMNAMES` file that is specified by Example 3-2.

*Example 3-2 DFSORT symbols specification*

```
//SYMNAMES DD *
POSITION,1
Honorific,*,16,CH
Surname,*,16,CH
/*
```

A non-descriptive DD name such as `HIPHOP` is not a good choice for splitting. It is better to use an extensible naming convention, such as the one that is shown in Example 3-3.

Use the **`DFSORT`** symbols capability for maintainable code. It allows you to map an input record and use the symbols in your **`DFSORT`** statements. In z/OS Version 2 Release 1 **`DFSORT`**, you can use the symbols in even more places in **`DFSORT`** statements.

In addition to **`INCLUDE`**, which specifies which records are kept, you can use **`OMIT`** to specify which records are not kept. Use whichever of these ways lead to simpler and more understandable **`DFSORT`** statements.

### Card dealing

In Example 3-3, **`SPLIT`** is used as a "card dealer", writing one record to each output DD, and then another to each, and so on.

*Example 3-3 DFSORT OUTFIL SPLIT as a card dealer*

```
OPTION COPY
OUTFIL FNAMES=(SPLIT01,SPLIT02,SPLIT03,SPLIT04),SPLIT
END
```

This approach is useful to ensure that each instance processes the same number of records.

### Chunking

Assume that you have four instances. If you want to pass the first 100,000 records to Instance 1, the next 100,000 to Instance 2, the third 100,000 to Instance 3, and the remainder to Instance 4, you can use statements similar to Example 3-4.

*Example 3-4 DFSORT OUTFIL chunking example*

```
OUTFIL FNAMES=(SPLIT01),STARTREC=1,ENDREC=100000
OUTFIL FNAMES=(SPLIT02),STARTREC=100001,ENDREC=200000
OUTFIL FNAMES=(SPLIT03),STARTREC=200001,ENDREC=300000
OUTFIL FNAMES=(SPLIT04),SAVE
```

In Example 3-4, the final **`OUTFIL`** statement uses **`SAVE`** to keep all the records that the other **`OUTFIL`** statements discarded. These **`OUTFIL`** statements collectively do not ensure that a further 100,000 records will be written to DD SPLIT04. If there are only 300,001 records in the input data set, only one is written to SPLIT04. If there are 1 million records in the input data set, SPLIT04 receives 700,000.

This technique might be useful if you want to allow instances to start in a staggered fashion. For example, the instance that processes the data in DD SPLIT01 in Example 3-4 might start before the instance that processes the data in DD SPLIT02.

### Ensuring that you do not lose records

Record selection criteria can become complex, particularly if more than one **OUTFIL INCLUDE** or **OMIT** statement is specified. In Example 3-4 on page 18, the final **OUTFIL** statement used the **SAVE** parameter.

To write code that simply and accurately specifies the records you want to keep that would otherwise be discarded, use **OUTFIL SAVE**. **SAVE** ensures that you write, to a data set, all the records that met none of the previous **OUTFIL INCLUDE** or **OMIT** criteria.

## 3.2.2 Checking data

One of the major sources of batch failures is bad input data. Here are two examples of where bad data can be input into a batch application:

- An online application that solicits user input without adequately checking the data the user enters.
- A data feed from another application or installation that contains bad data.

It is better to fix application problems before re-engineering for parallelism, whether by splitting multi-step jobs or using the techniques in this publication. However, sometimes this situation is no realistic. The fan-out job is a good place to include logic to check input data:

- The fan-out job passes over the input data anyway. Additional processing to check data can be inexpensive if conducted in this pass.
- If the application also has a step that passes over the input data and checks for errors, adding split processing is unlikely to add much processing cost.

Example 3-5 is a simple example of how you might detect bad data. Much more sophisticated checking is feasible. The example shows some **DFSORT** logic to both split an input file and check a field contains valid data.

*Example 3-5 Sample DFSORT statements for a combined split and data check*

```
OPTION COPY
OUTFIL FNAMES=(BADFORMT),INCLUDE=(21,8,ZD,NE,NUM,OR,31,5,PD,NE,NUM)
OUTFIL FNAMES=(BADVALUE),OMIT=(31,5,PD,EQ,+1,OR,31,5,PD,EQ,+2)
OUTFIL FNAMES=(SLICE1),INCLUDE=(31,5,PD,EQ,+1)
OUTFIL FNAMES=(SLICE2),SAVE
END
```

The first **OUTFIL** statement checks that both a zoned decimal field and a packed decimal field contain valid data according to their specified data formats. The records that fail these checks are written to DD BADFORMT (for “bad format”).

The second **OUTFIL** statement checks that the packed decimal field contains either 1 or 2. The records that contain different values in the field are written to DD BADVALUE (for “bad value”).

The third and fourth **OUTFIL** statements split valid records according to the value in the same packed decimal field that the first two **OUTFIL** statements check.

Here are three ways to check whether there were any bad records:

- Use the **ICETOOL COUNT** operator with the **NOTEMPTY** parameter to set a return code of 12 if the BADFORMT data set contains records. Repeat with the BADVALUE data set.
- Parse or inspect the output messages for the BADFORMT and BADVALUE DD statements in the **DFSORT SYSOUT** using Tivoli Workload Scheduler's Job Completion Checker.
- Use **IDCAMS REPRO**, as shown in Example 3-6. In this example, an empty CHECKME data set causes **IDCAMS** to terminate with return code 0, and one with records in it terminates with return code 8.

*Example 3-6 Using REPRO to check for bad records*

```
//IDCAMS   EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//CHECKME  DD DISP=SHR,DSN=<your file>
//DUMPME   DD SYSOUT=*
//SYSIN    DD *
 REPRO INFILE(CHECKME) OUTFILE(DUMPME) COUNT(1)
 IF LASTCC = 0 THEN SET MAXCC = 8
 ELSE SET MAXCC = 0
/*
```

## 3.3 Replicated job instances

In our example, the replicated job instances process their data as though it is not split. As described in 3.2, "Fan-out" on page 16, it is sometimes possible to allow each instance to process all the data but add logic to discard most of it. In this section, it is assumed a fan-out job created subsets of the input data, with each instance processing one subset.

Most of the design decisions for the repeated job instances are data management decisions. These decisions are described in Chapter 2, "Data management considerations" on page 9.

Replicated job instances look much like the original job, but perhaps are simplified:

- Some record selection logic might be removed.

  It can be specified in the fan-out job, as described in 3.2, "Fan-out" on page 16.
- If the original job wrote a report, the logic to format the report can be removed.

  If the original job wrote a report, you must modify the program's logic to write a data set for the fan-in job to merge for the reporting job, as described in 3.5, "Reporting" on page 21.

## 3.4 Fan-in

If the original job writes output data, the split jobs might write subsets of that data. This data must be merged to reproduce the data that the original job wrote. A similar consideration is reproducing any reports that the original job created. This topic is described in 3.5, "Reporting" on page 21.

After all the instances have run, a separate job to collate the data must run. In this publication, this job is called the *fan-in* job. Whether you separate reporting from fan-in, reporting usually is separated from the split jobs. This is untrue if each instance processes the data to create a single report. For example, in 3.2, “Fan-out” on page 16, you might select a technique that splits data into regions, and the reports that the original job produced were region by region. In that case, each instance can produce a report.

## 3.5 Reporting

Many jobs produce reports or similar output. Here are two examples:

- A summary of the run. One example is sales figures by region.
- Output data to send to printers to print a batch of account statements.

You often can combine reporting with the fan-in stage that is described in 3.3, “Replicated job instances” on page 20, but there are advantages in keeping these steps separate:

- You can reuse the raw data for the reports for other purposes.
- You can modernize the format of the report, as described in Appendix A, “Application modernization opportunities” on page 101.
- You can check whether the data is clean before you create the report.
- You can simplify the fan-in program.
- You can write the report in a more modern style, perhaps using a report generator.

Many existing jobs that create reports or output data have the reporting logic embedded in the main program. Changing applications introduces the risk of errors, so consider whether separating out reporting is worth the risk.

The process of creating a report is composed of two separate logical stages:

- Report data collation
- Report formatting

Although these stages are combined in many production programs, it is useful to logically separate them. Indeed, physically separating them might allow the report data collation stage to be combined with the fan-in stage that is described in 3.4, “Fan-in” on page 20.

### 3.5.1 Report data collation

Report data collation is the process of marshalling data into a form where it can be formatted into a report, as described in 3.5.2, “Report formatting” on page 22. For example, a job might produce total sales. In the original job, the raw sales number for each region was created as part of mainline processing. In our example, a data set containing a list of regions and the related numbers is created.

Example 3-7 shows such a data set, but with the numbers formatted for human readability. In reality, they are stored in a binary format, such as packed decimal. Also, blanks are replaced by underscores.

*Example 3-7 A collated data file*

```
North___4000
South___3210
East____2102
West____3798
```

Report data collation is made more complex when a job is split. For example, the split version of the data set might be what is shown in Example 3-8 and Example 3-9.

*Example 3-8 First instance's output data*

```
North___2103
South___2110
East____1027
West____2001
```

*Example 3-9 Second instance's output data*

```
North___1897
South___1100
East____1075
West____1797
```

In this case, it is simple to reproduce the data set in Example 3-7 by matching the rows in the two data sets based on the Region field and summing the numerical values. In this case, the split jobs can run the same way as the original job, if it already produced a report data file.

Slightly more complex than this summation case is averaging. To create a properly calculated average, complete the following steps:

1. Make each split job create a report data file with both the total and the count in.
2. Merge the rows in these files, again based on a key, and sum the totals and counts.
3. Divide the totals by the counts to derive averages.

Both summation and averaging are relatively simply, but the latter requires keeping different data in the split jobs' report data files.

Other types of reporting data require more sophisticated marshalling techniques. In some cases, marshalling the data from split job instances might not be possible.

## 3.5.2 Report formatting

Report formatting is the process of taking the collated report data, as described in 3.5.1, “Report data collation” on page 21, and creating a readable report, or an externally processable file (such as for check printing). Report formatting is insensitive to whether the job is split, if report data collation is designed correctly. Some of the reporting logic can be taken from the original job, but it must be adapted to process the collated report data file.

It is beyond the scope of this publication to describe reporting techniques in detail, but here are some basic considerations:

- Consumers of the original reports expect to see their reports unchanged, which usually is possible.
- New reporting tools and techniques might be used to make the reports more attractive. Consider using techniques to the ones that are described in Appendix A, “Application modernization opportunities” on page 101.
- Some of the reporting might be unnecessary. In many applications, the need for the original reports has disappeared. Consider removing reports from the application.

# 4

# Considerations for job streams

Previous chapters have focused on splitting individual jobs. This chapter describes some selected techniques for managing the splitting of entire job streams.

## 4.1 Scenarios

Few batch applications make up a single batch job. Most consist of a network of related jobs with dependencies between them. Some batch splitting scenarios require only one job to be split. Other scenarios benefit from more than one job being split.

Here are some common scenarios involving multiple jobs:

1. Splitting all the jobs in a stream at once.

   Splitting all the jobs is an ambitious approach but might be simpler to implement.

2. Splitting a subset of the jobs.

   In many cases, only a few jobs need splitting; the rest do not run long enough to warrant splitting.

3. Splitting a single job, then refining the implementation by splitting another, and so on.

   Gradually extending the scope of splitting is a cautious approach.

There are many multi-job scenarios, of which the above are just a few examples. All the scenarios have common issues to resolve. The remainder of this chapter describes some of these issues.

## 4.2 Degrees of parallelism

Neighboring jobs that are split with different degrees of parallelism might be problematic. One problem might be marshalling data between instances of the two jobs, as described in 4.3, “Data marshalling” on page 26.

For example, if JOBA is split into two instances (JOBA1 and JOBA2), but its immediate successor, JOBB, is split into three instances (JOBB1, JOBB2, and JOBB3), managing the different degrees of parallelism can be difficult. For JOBA, the degree is two, and for JOBB, it is three.

It is best to standardize where possible. In this example, standardize with either two or three.

Standardization is more difficult when the two jobs perform best with different degrees of parallelism. Consider making one job’s degree of parallelism a multiple of the other.

For example, JOBC might perform best with a degree of parallelism of eight and JOBD with a degree of parallelism of 20. In this case, consider of 16.

## 4.3 Data marshalling

In this publication, data marshalling refers to moving data from one job step to another one. The term encompasses a wide range of issues.

For example, 4.2, “Degrees of parallelism” on page 26 describes the case of a 2-way split JOBA and a 3-way split JOBB. JOBB depends on JOBA. Suppose that this dependency is caused by JOBA writing a data set that JOBB reads. When split, the instances of JOBA must write data in a manner that the three instances of JOBB can process.

It is easier if JOBA and JOBB were each split into the same number of instances. In the case of two instances, each JOBA1 might write data that JOBB1 reads and JOBA2 data that JOBB2 reads. Marshalling data this way might be impossible. It requires a good understanding of both jobs.

Another possibility is changing each JOBA to write three data sets, one each for JOBB1, JOBB2, and JOBB3. JOBA1 writes three data sets and so does JOBA2. Again, this approach might not be feasible.

### 4.3.1 Unnecessary fan-in and subsequent fan-out jobs

Suppose that JOBG is split into four instances and JOBH into four instances. In our example, the instances of JOBG are preceded by a fan-out job, JOBFO1, and followed by a fan-in job, JOBFI1. Similarly, instances of JOBH are preceded by JOBFO2 and followed by JOBFI2.

Are the intermediate fan-in job (JOBFI1) and fan-out job (JOBFO2) necessary? A good understanding of the application is needed to find out.

Scenario 3 in 4.1, “Scenarios” on page 26 is a case where piecewise implementation of job splitting can lead to unnecessary fan-in and fan-out jobs: one job is split now, and then another later on.

Are these intermediate jobs harmful? If they process much data, they might delay the application. If it is only a little data, the delay might be negligible.

## 4.4 Sequences of jobs

If you plan to implement job splitting for a sequence of jobs, consider the order in which you implement it. You might be able to avoid some of the issues that are described in 4.3, “Data marshalling” on page 26 if you follow one of the following approaches to implementing job splitting.

1. Split all the jobs at once, rather than in stages.

   This is Scenario 1 in 4.1, “Scenarios” on page 26.

2. Extend your sequence of split jobs by splitting the ones that are just before or just after the sequence.

   For example, split the job that immediately follows the last split job in the sequence.

Splitting all or many of the jobs at once might produce better results because of the following reasons:

- You can test everything at once.
- You can perform optimizations that are not apparent when splitting just a few jobs.

## 4.5 Scheduling

Job scheduling is described in Chapter 5, “Operational considerations” on page 29.

5

# Operational considerations

This chapter describes operational considerations when splitting batch jobs. These usually are the responsibility of the operations or scheduling teams. They include elements that are related to JCL, scheduling, and issues that are related to the usage of both. JCL elements are written with consideration of z/OS features for Version 2.1. Scheduling elements are, where possible, described without using product terminology. But where practical, examples are given, which are written with consideration of IBM Tivoli Workload Scheduler for z/OS features for Version 9.1.

Here are key terms that are used in this chapter:

- Scheduling object: A generic term to cover all scheduling products. It refers to the object that contains the planning rules that define when a set of jobs run and in what order. In Tivoli Workload Scheduler for z/OS terms, this is known as an *application definition*.
- Operation: An individual item of a workload within the scheduling object. In most cases, this is a job, but it can be other things, including a dummy process that exists for consolidating dependencies.
- Planning rules: The rules that define when a scheduling object is run. In Tivoli Workload Scheduler for z/OS terms, these are known as *runcycles*.

**Note:** Some examples in this chapter use an assembly language implementation of the Job-Library-Read exit (EQQUX002) and a REXX toolkit for the Tivoli Workload Scheduler for z/OS Program Interface (PIF) known as the Scheduling Operational Environment (SOE). Both tools can be downloaded from the IBM developerWorks® z/Glue community. Now, they are not formally part of the product. As such, they are supported only by the community at the z/Glue website:

`https://www.ibm.com/developerworks/community/groups/community/zGlue`

In each case, where these tools are referenced, an outline of the process taking place is given, so that it is possible to create custom solutions from scratch by using the published exit points and the documented program interface.

**Note:** For supplementary information for this chapter, and code samples, see Appendix B, "Operational supplement" on page 107.

## 5.1 Preparing for parallel batch processing

To take an individual batch job and split it into multiple instances requires little forethought as a one-off task. But as the number of jobs being split into multiple instances increases, advanced planning is important. Careful planning before progressing towards a parallel batch strategy pays dividends later.

### 5.1.1 Key components of a parallel batch structure

When dealing with parallel batch scheduling, there are a few key components to consider that help the design of the operational elements that are needed to run the workload. Not all of the components apply to every scenario, but following this model at the design stage ensures that you consider the important elements.

Figure 5-1 shows the key components of a parallel batch structure.

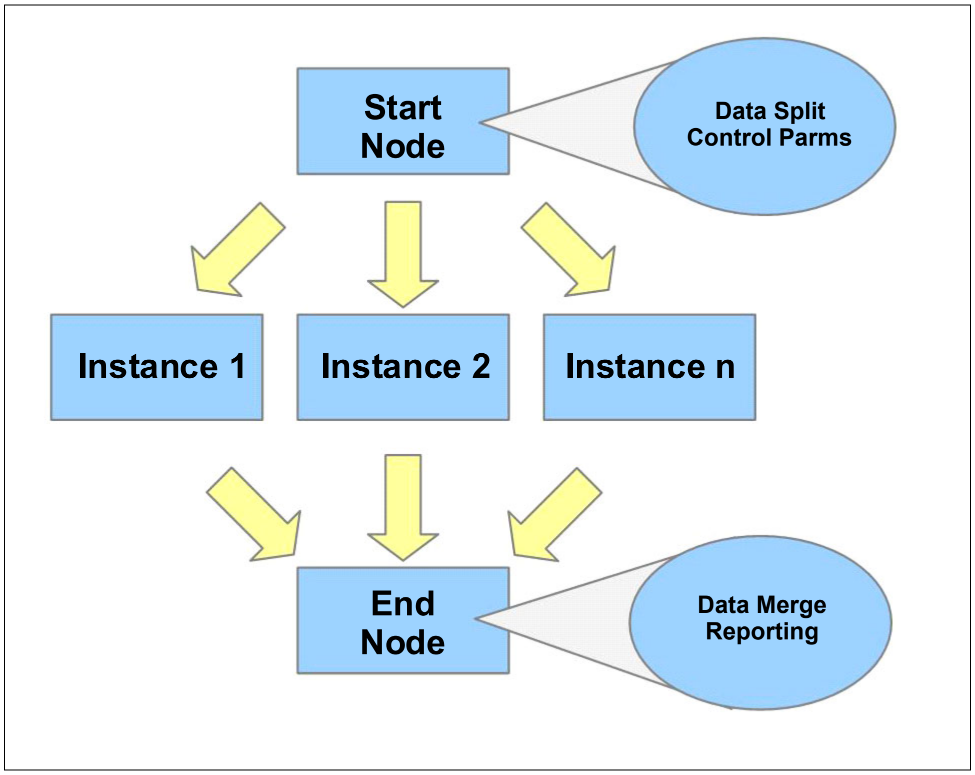


*Figure 5-1 Key components of a parallel batch structure*

#### Start node

The start node is the point of origin of the process. At an absolute minimum, it should be the focal point of all dependencies for the parallel instances, fanning out dependencies to each instance. So, each instance depends on the start node, and the start node can hold dependencies for any prerequisites for the entire process.

Depending upon the nature of the original process being split, there might be a need to split input data into separate data sets. If so, then the split process should be included within the start node. For examples about how to split data, see 3.2.1, “DFSORT OUTFIL record selection capabilities” on page 17.

For the individual job instances to know the scope of the data that they are expected to process, they might need specific control statements or program arguments. For static splitting of jobs, the control statements can be set up once when the job is first split, and each instance processes data within the same scope every run. However, with dynamic splitting, there must be a process to generate the control statements or arguments of each run, which is based upon the criteria that is used to split the processing. The job to generate such control information is included within the start node. For examples about how to pass control information from the start node to each individual parallel instance, see 5.2.4, “Control data communication” on page 39.

### Parallel instances

After any processing takes place in the start node, each of the individual parallel instances can run. Each depends on the start node and runs almost identical JCL, with the only differences being job name, control information, and any instance-specific data sets. Section 5.2.3, “Minimizing JCL maintenance” on page 38 provides guidance about how to write generic JCL.

The parallel instance jobs are the part of the process that performs the business function.

### End node

The end node is the endpoint at which all the parallel processing converges after it completes. It depends on each of the parallel instances, so it becomes the focal point for any processing that must depend on the parallel processing as a whole.

In some cases, each parallel instance can produce output data that must be merged. A job to perform this kind of processing is included in the end node. For examples of how to do this task, see 5.2.6, “Merging output data” on page 47.

Depending on the nature of the processing that is taking place, aggregate reporting might be needed, rather than individual reporting within each instance. Such reporting is included within the end node, and might also need to depend on a data merge process.

### Splitting multiple jobs for parallel processing

The start node, instances, and end node model is a simplistic approach for splitting one single job into multiple instances. In cases where more than one job in succession is being split into parallel instances, then a slightly different approach should be employed.

With parallelism, there potentially are two data streams that can come out of each instance:

- Full data: This is an unsummarized data stream that is destined as input for an equivalent instance of a successor parallel job.
- Summary data: This is a smaller summarized data stream, which is destined to be merged and processed in a single “end node” job.

It is not efficient use of processing to combine the “full data” stream back into a single data source, bring all instances to a single dependency, and then split the data again for a successive job. Instead, a better approach is allowing the full data to pass as output from each parallel instance of one split job into the corresponding instance of a successor split job. Each successor instance directly depends on its corresponding predecessor.

Figure 5-2 shows how each successive split job can have corresponding instances that depend on each other, with each end node's processing depending on the all the instances of the relevant job. So, the full data travels into the next instance and the summary data travels into an end node process.

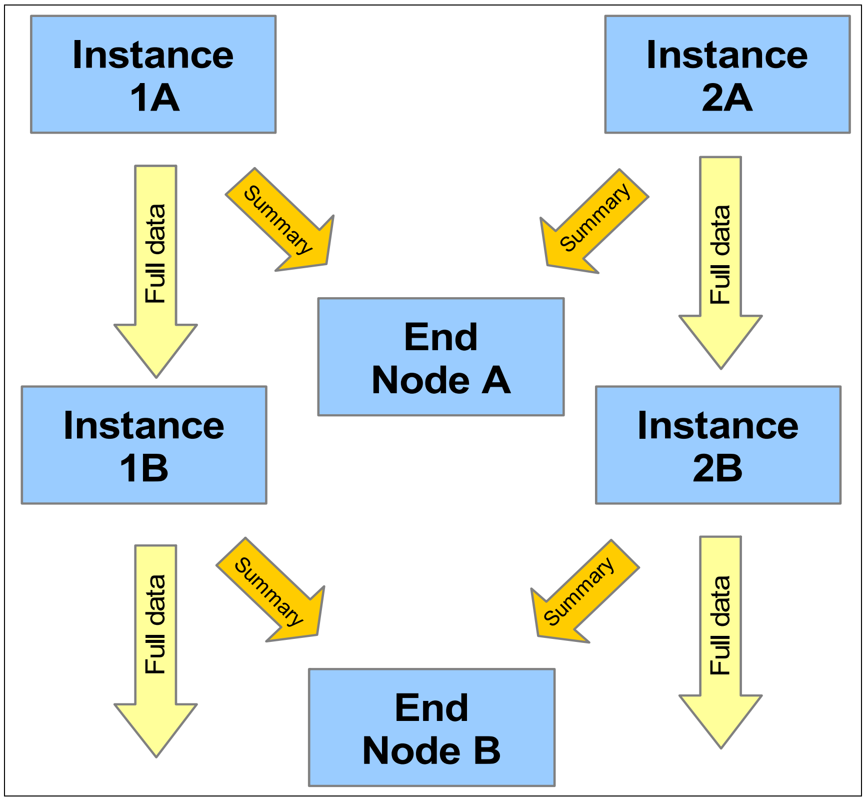


*Figure 5-2 Multiple successive job splits*

## 5.1.2 Naming considerations

The most important operational start point is a good naming convention for job names, data set names, and scheduling elements.

### Job naming convention

In the context of parallel batch, a job naming convention must serve the following purposes.

- To run the same process multiple times in parallel, a unique name is required for each instance. Otherwise, the Job Entry Subsystem (JES) forces all jobs with the same name into a single stream. JES3 has an option DUPJOBNM that can be turned on to allow multiple jobs to run with the same name simultaneously.
- To allow all jobs that form the parallel processing to be easily identified, grouped, and monitored.
- To communicate to the content of the JCL which instance of the parallel process is being run.

In most sites, a job naming convention is in place to cover elements such as phases in the development lifecycle, business application, geographic location, and run frequency, among others. This convention typically determines the first three or four characters of the job name. Any convention for parallel instance identification must also fit within the existing site conventions.

As a preferred practice, allocate a single character in the job name to identify the instance. To make it simple to find all instances of the same job, assigning the last character of the job name causes all parallel instances of a job to group when sorted by job name.

Example 5-1 shows an example of how a naming convention can be formed to handle parallel instances.

*Example 5-1 Parallel job naming convention*

```
<p><aa><r><fff><i>

The components are composed of the following elements.
<p> = Phase of the development lifecycle
<aa> = Business application
<r> = Run frequency
<fff> = Job function
<i> = Parallel instance
```

For example, PPYMUPDA is the first instance of the Production Payroll monthly update job and PPYMUPDB are the second instance.

Using a letter of the alphabet as the initial choice for an instance is preferable, as it allows for more splits before having to switch conventions. For example, with letters of the alphabet, you can allow for 2-up (A - B), 4-up (A - D), 8-up (A - H), and 16-up (A - P). When numbers are used, the highest split that is allowed is 8-up (1 - 8). To handle a 32-up split, A - Z followed by 0 - 5 can be employed as a convention.

**Note:** If you want to create generic JCL that automatically can determine which instance is running, then it is important that the instance character is in a fixed position in the job name. This is not an issue with the convention in Example 5-1 on page 33 if job names are always eight characters, but if the *Job function* can be of variable length, there is no guarantee that *Parallel instance* always will be character 8. In these cases, ensure that the instance identifier is positioned before any variable function, for example, <p><aa><r><i><fff>.

## Data set naming convention

The naming convention for data sets serves the following purposes:

- To allow unique control and data streams to be passed into and out of each instance.
- To group parallel data streams in a way that allow automated aggregation of data sets.

For the most consistent and flexible JCL, the same single character that is used for the job instance should be used as part of the data set name. There should also be a grouping identifier so that you can quickly find all inputs and outputs to a stream.

For example, `PPY.MUPDA.INPUT` and `PPY.MUPDA.OUTPUT` are the input and output data sets for the Production Payroll monthly update instance A, so `PPY.MUPDB.INPUT` and `PPY.MUPDB.OUTPUT` are the equivalent for instance B.

There are some potential pitfalls to avoid with a data set naming convention.

- The low-level qualifier (LLQ) should be avoided for identifying the owning instance. Suffixing the end of the data set name with the instance character might seem appropriate to match the job naming convention. However, this can conflict with Data Facilities System Managed Storage (DFSMS). It is a preferred practice to use the LLQ of data sets to drive automated class selection (ACS) routines to decide such things as Management Class or Data Class of the data sets. By suffixing the instance name to the LLQ, the ACS routines might also require amendment.
- The instance name should never be used by itself as a qualifier of the data set name. Although this works if instances are restricted to alphabetic characters, if at some point a 16-up split was increased to 32-up, any numeric instances cause JCL errors. For example, PPY.MUPD.A.INPUT is alright, but PPY.MUPD.3.INPUT is not.

### Scheduling object naming convention

For scheduling objects, the need for a naming convention is conditional on both the scale of the split taking place and the techniques that are engaged because small-scale static parallelism most likely will be written with a single scheduling object containing all the instances within.

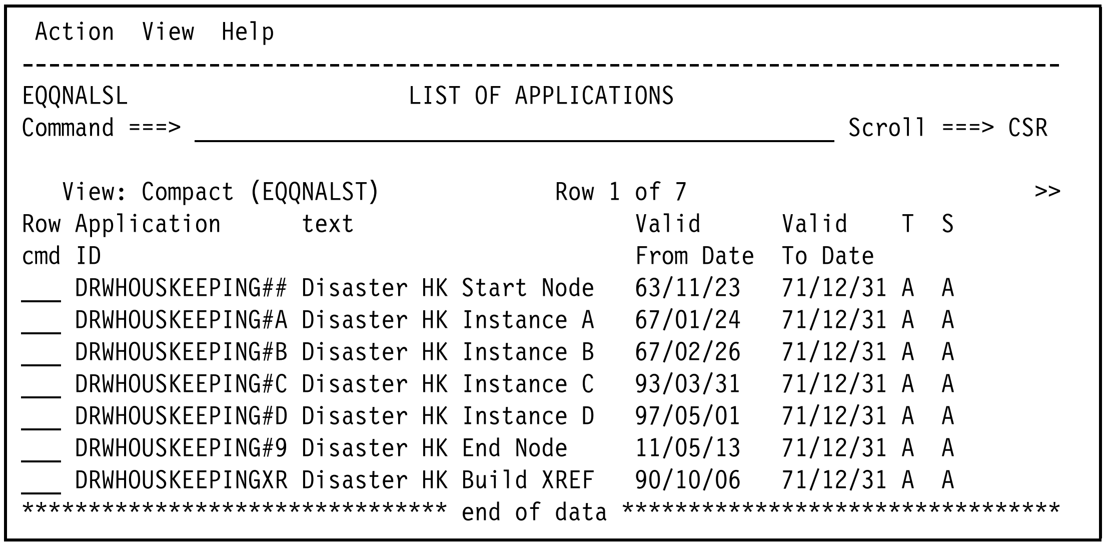
If scheduling is set up so that each instance is contained within a separate scheduling object, then the following requirements are placed upon a naming convention.

- To run the same process multiple times in parallel, a unique name is recommended for each instance. Although scheduling products might allow an object of the same name many times, this is often at the expense of clarity and ease of resolving dependencies.
- Allow all scheduling objects that form the parallel processing to be easily identified, grouped, and monitored.

The same instance character should be used in the scheduling object name, to clearly tie job names, data set names, and scheduling objects together.

Depending on the product that is being used, the scheduling object name is often longer than eight characters. In those cases, it is worth considering designating particular character positions in the name. For example, you can distinguish parallel batch from non-parallel batch and identify start node, end node, and instances. This allows elements of a parallel batch structure to easily be identified.

Finding a convention that does not require a full retrofit of schedule object names might be a challenge. Figure 5-3 on page 35 shows an example of how this might be done within a Tivoli Workload Scheduler for z/OS database.



```
 Action  View  Help
------------------------------------------------------------------------------
EQQNALSL                     LIST OF APPLICATIONS
Command ===> ________________________________________________ Scroll ===> CSR

   View: Compact (EQQNALST)            Row 1 of 7                          >>
Row Application      text                    Valid      Valid    T  S
cmd ID                                       From Date  To Date
___ DRWHOUSKEEPING## Disaster HK Start Node  63/11/23   71/12/31 A  A
___ DRWHOUSKEEPING#A Disaster HK Instance A  67/01/24   71/12/31 A  A
___ DRWHOUSKEEPING#B Disaster HK Instance B  67/02/26   71/12/31 A  A
___ DRWHOUSKEEPING#C Disaster HK Instance C  93/03/31   71/12/31 A  A
___ DRWHOUSKEEPING#D Disaster HK Instance D  97/05/01   71/12/31 A  A
___ DRWHOUSKEEPING#9 Disaster HK End Node    11/05/13   71/12/31 A  A
___ DRWHOUSKEEPINGXR Disaster HK Build XREF  90/10/06   71/12/31 A  A
******************************** end of data *********************************
```

*Figure 5-3 A cohabiting naming convention*

The intent behind such a convention is to find a character position in the object name with which you can tag parallel batch scheduling objects by using a special character that does not exist at that position in the pre-existing workload. So, an existing workload does not need to be renamed, or at best only a minimal amount, and any new parallel batch scheduling objects can be clearly tagged with this special character, which then allows you to designate other character positions as having special meanings *only* for parallel batch elements, again without conflicting with existing elements in the workload.

In the example in Figure 5-3, character position 15 is designated as the tag point for parallel batch elements. So, `DRWHOUSKEEPINGXR` appears to not be a parallel batch object, and all the others are.

Then, for the tagged scheduling objects, character 16 has special significance:

- # = Start node of a parallel batch structure
- 9 = End node of a parallel batch structure
- @ = Self contained parallel batch structure including Start, Instances, and End (not shown)
- A-Z or 0-5 = up to 32 parallel instances

The example shows a start node, instances, and end node all kept together in sequence.

Although this particular convention might conflict with existing workload at some sites, the concept of finding a special tag position and character should allow a similar convention to be designed and adopted with minimal retrofitting effort.

## 5.2 JCL splitting techniques

Starting from an original single instance of a job, to run the same process multiple times in parallel requires some considerations regarding job name and data must be engineered into the JCL. Various steps can be taken to split a job for parallel processing.

## 5.2.1 Base JCL

Figure 5-4 shows the original job before any amendments are made. In this example, we have the Disaster Recovery Weekly Housekeeping process, which runs three steps.

- CLEANUP is a preemptive deletion of the output file for step PROCESS. This allows the job to be rerun from the top with no additional recovery processes.
- AUDIT is a tracking step that records the keyrange of the records that are being processed by each job that runs.
- PROCESS is driven by an input file and produces an output file.

```
//DRWHOUPD JOB (99999),'DR WHOUSEKP',USER=MCGANN,
//         CLASS=A,MSGCLASS=X
//*
//CLEANUP  EXEC PGM=IEFBR14
//DD1      DD DISP=(MOD,DELETE),DSN=DR.WHOUSE.OUTFILE,
//         SPACE=(TRK,1)
//*
//AUDIT    EXEC PGM=DRTRACK,
//         PARM='DRWHOUPD 000001 999999'
//STEPLIB  DD DISP=SHR,DSN=DR.LOADLIB
//TRACKLOG DD DISP=SHR,DSN=DR.WOTAN.KSDS
//*
//PROCESS  EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB  DD DISP=SHR,DSN=DR.LOADLIB
//SYSOUT   DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//INPFIL01 DD DISP=SHR,DSN=DR.WHOUSE.INFILE
//INPFIL02 DD DISP=SHR,DSN=DR.MATRIX.KSDS
//OUTFIL01 DD DISP=(NEW,CATLG,KEEP),DSN=DR.WHOUSE.OUTFILE,
//            SPACE=(TRK,(10,10)),RECFM=FB,LRECL=101
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
  DSN SYSTEM(DRVW)
  RUN PROGRAM(GLXY4K9) PLAN(GLXY4K9)
  END
/*
//
```

*Figure 5-4 Base JCL before being split*

## 5.2.2 Simple JCL splitting

Figure 5-5 on page 37 shows a simple first level of JCL splitting. The following changes were made:

- The job name was amended to make the last character the instance name. In this example, A is shown.
- A JCL symbolic, TINST, was created to represent the parallel track instance character. This is set to match the last character in the job name.
- The input and output data set names were amended to include the TINST symbolic as part of the name, allowing each instance to have unique input and output data sets.

- Two additional JCL symbolic parameters, MIN and MAX, were created to hold the start and end of the key ranges being updated by this job.
- The PARM for the AUDIT step was amended to include the TINST, MIN, and MAX symbolic parameters.

The result of these changes is that it is now possible to make multiple copies of this job with only minimal changes to the job name and the values on the **SET** statement at the head of the job.

```
//DRWHOUPA JOB (99999),'DR WHO A',USER=MCCOY,
//         CLASS=A,MSGCLASS=X
//*
//         SET TINST=A,MIN=000001,MAX=250000
//*
//CLEANUP  EXEC PGM=IEFBR14
//DD1      DD DISP=(MOD,DELETE),DSN=DR.WHO&TINST..OUTFILE,
//         SPACE=(TRK,1)
//*
//AUDIT    EXEC PGM=DRTRACK,
//         PARM='DRWHOUP&TINST &MIN &MAX'
//STEPLIB  DD DISP=SHR,DSN=DR.LOADLIB
//TRACKLOG DD DISP=SHR,DSN=DR.VONWER.KSDS
//*
//PROCESS  EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB  DD DISP=SHR,DSN=DR.LOADLIB
//SYSOUT   DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//INPFIL01 DD DISP=SHR,DSN=DR.WHO&TINST..INFILE
//INPFIL02 DD DISP=SHR,DSN=DR.MATRIX.KSDS
//OUTFIL01 DD DISP=(NEW,CATLG,KEEP),DSN=DR.WHO&TINST..OUTFILE,
//            SPACE=(TRK,(10,10)),RECFM=FB,LRECL=101
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
  DSN SYSTEM(DRVW)
  RUN PROGRAM(GLXY4K9) PLAN(GLXY4K9)
  END
/*
//
```

*Figure 5-5 A simple first-level split*

These slightly amended copies can then be run in parallel without clashes in terms of JCL. The TINST value is submitted into the input and output data set names, allowing separate data feeds. Example 5-2 shows how the input file is resolved to give a unique name per instance.

*Example 5-2 TINST substitution in action*

```
14 //INPFIL01 DD DISP=SHR,DSN=DR.WHO&TINST..INFILE
   IEFC653I SUBSTITUTION JCL - DISP=SHR,DSN=DR.WHOA.INFILE
```

## 5.2.3 Minimizing JCL maintenance

Although only minor changes are needed for each instance, JCL of this style results in the core application JCL being duplicated within each job member. So, as the number of instances increases, so does the maintenance impact if any changes must be made to the core application JCL.

### Cataloged procedures

To remove the impact of duplicated application JCL, convert the job into a cataloged procedure. Figure 5-6 shows the job after converting it to a cataloged procedure.

```
//DRWHOUSE PROC TINST=*,  /* INSTANCE IDENTIFIER   */
//         MIN=*,         /* LOWER AUDIT KEY RANGE */
//         MAX=*          /* UPPER AUDIT KEY RANGE */
//*
//CLEANUP  EXEC PGM=IEFBR14
//DD1      DD DISP=(MOD,DELETE),DSN=DR.WHO&TINST..OUTFILE,
//         SPACE=(TRK,1)
//*
//AUDIT    EXEC PGM=DRTRACK,
//         PARM='DRWHOUP&TINST &MIN &MAX'
//STEPLIB  DD DISP=SHR,DSN=DR.LOADLIB
//TRACKLOG DD DISP=SHR,DSN=DR.VONWER.KSDS
//*
//PROCESS  EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB  DD DISP=SHR,DSN=DR.LOADLIB
//SYSOUT   DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//INPFIL01 DD DISP=SHR,DSN=DR.WHO&TINST..INFILE
//INPFIL02 DD DISP=SHR,DSN=DR.MATRIX.KSDS
//OUTFIL01 DD DISP=(NEW,CATLG,KEEP),DSN=DR.WHO&TINST..OUTFILE,
//            SPACE=(TRK,(10,10)),RECFM=FB,LRECL=101
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
  DSN SYSTEM(DRVW)
  RUN PROGRAM(GLXY4K9) PLAN(GLXY4K9)
  END
/*
```

*Figure 5-6 Core application JCL converted to a cataloged procedure*

The changes that are required to turn the job into a procedure are minimal:

- The JOB card is removed.
- The SET statement is converted to a **PROC** statement.
- The job end statement is removed (//).

The result is that the JCL that runs in each job instance is reduced, as shown in Figure 5-7 on page 39.

```
//DRWHOUPA JOB (99999),'DR WHO A',USER=PERTWEE,
//         CLASS=A,MSGCLASS=X
//*
//DRWHOUSE EXEC DRWHOUSE,TINST=A,MIN=000001,MAX=250000
//
```

*Figure 5-7 Simple running JCL*

## 5.2.4 Control data communication

Even with cataloged procedures, each individual piece of running JCL is written to pass in control information, so each instance can be informed of the range of data it is required to process. There are techniques that can be performed to allow this information to be passed through to the underlying programs with minimal or no differences between each JCL instance.

### JCL tailoring directives and variables

Although JCL itself does not provide mechanisms to set JCL symbols that can reflect the job name or portions, Tivoli Workload Scheduler for z/OS has many JCL tailoring directives. These directives are referred to as *OPC directives*, which allow JCL to be dynamically formed at submission time. Tivoli Workload Scheduler for z/OS also has many supplied variables that aid in writing dynamic JCL, including the job name.

Figure 5-8 shows some of the directives and variables that are used to remove some of the unique elements within the JCL job.

```
//*%OPC SCAN
//&OJOBNAME JOB (99999),'DR W HKP',USER=BAKER,
//         CLASS=A,MSGCLASS=X
//*%OPC SETVAR TINST=SUBSTR(&OJOBNAME,8,1)
//*
//DRWHOUSE EXEC DRWHOUSE,TINST=&TINST,MIN=000001,MAX=250000
//
```

*Figure 5-8 Supplied variables and substrings*

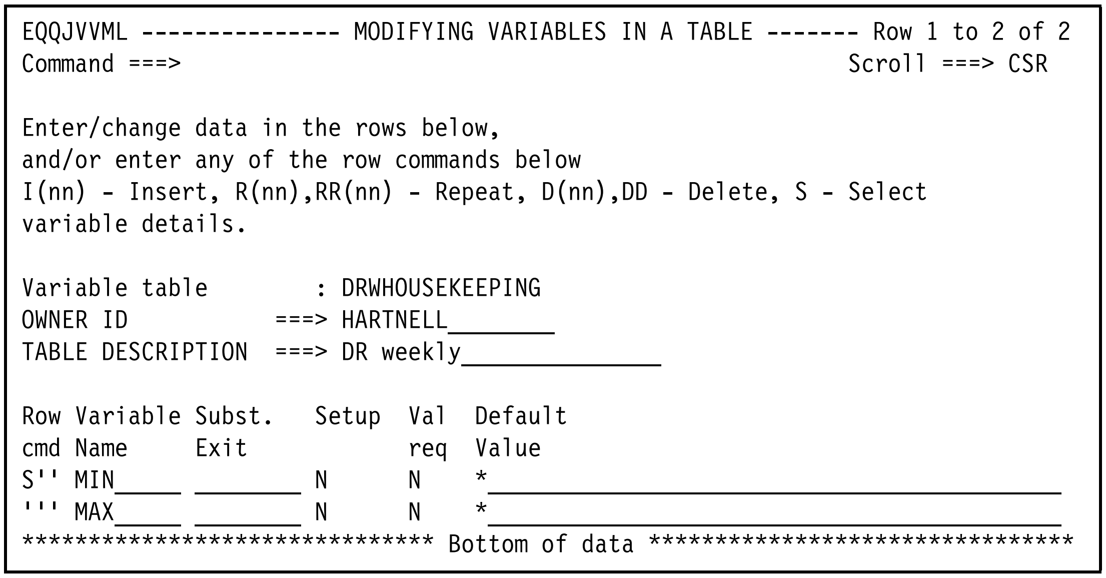
The OPC SCAN directive tells Tivoli Workload Scheduler for z/OS to scan the remainder of the job for further OPC directives, or JCL variables. The job name is replaced with `&OJOBNAME`, which always resolves to the name of the job being scheduled at submission time. This removes the need to change individually the job name for each extra instance that you make of the job. The **`OPC SETVAR`** directive can then take a substring of the **`&OJOBNAME`** variable to extract the instance character from the job name. Figure 5-9 shows how this JCL is resolved by Tivoli Workload Scheduler for z/OS.

```
//*>OPC SCAN
//DRWHOUPA JOB (99999),'DR W HKP',USER=BAKER,
//         CLASS=A,MSGCLASS=X
//*>OPC SETVAR TINST=SUBSTR(DRWHOUPA,8,1)
//*
//DRWHOUSE EXEC DRWHOUSE,TINST=A,MIN=000001,MAX=250000
//
```

*Figure 5-9 Resolution of supplied variables and substring*

This still leaves the MIN and MAX symbolic parameters as being different for each instance of the job. This difference can be resolved by using a Tivoli Workload Scheduler for z/OS feature that is known as *dependent variables*. This feature allows you to create a variable table that can return different values for each variable, depending upon the value of another variable. Because the OPC SETVAR directive created the TINST variable that contains the instance character, this variable can be used to drive the values of MIN and MAX.

Figure 5-10 shows the modification of a JCL variable table (option 1.9.2). Having added the two variables MIN and MAX as normal, you can select each variable individually to set dependencies.



```
EQQJVVML --------------- MODIFYING VARIABLES IN A TABLE ------- Row 1 to 2 of 2
Command ===>                                                  Scroll ===> CSR

Enter/change data in the rows below,
and/or enter any of the row commands below
I(nn) - Insert, R(nn),RR(nn) - Repeat, D(nn),DD - Delete, S - Select
variable details.

Variable table        : DRWHOUSEKEEPING
OWNER ID           ===> HARTNELL________
TABLE DESCRIPTION  ===> DR weekly_______________

Row Variable Subst.   Setup  Val  Default
cmd Name     Exit            req  Value
S'' MIN_____ ________ N      N    *__________________________________________
''' MAX_____ ________ N      N    *__________________________________________
******************************* Bottom of data ********************************
```

*Figure 5-10 Creating a variable table*

After you select an individual variable, the MODIFYING A JCL VARIABLE panel opens (EQQJVVMP). Nothing on this panel requires amendment, so enter **DEP** to open the JCL VARIABLE DEPENDENCY VALUE LIST panel (EQQJVDVL), as shown in Figure 5-11 on page 41. From here, it is possible to set up the dependency between TINST and MIN.

```
EQQJVDVL ------------ JCL VARIABLE DEPENDENCY VALUE LIST ----- Row 1 to 2 of 4
Command ===>                                                  Scroll ===> CSR

Enter any of the row commands below:
I(nn) - Insert, R(nn), RR(nn) - Repeat, D(nn) - Delete
Enter DEL   command to delete this dependency.

Dependent variable    : MIN
Variable table        : DRWHOUSEKEEPING   DR weekly
Default value         : *

INDEPENDENT                  SUBSTRING              SUBSTRING
VARIABLE    ===> TINST       START     ===> 00      LENGTH    ===> 00

Row
cmd
''' VALUE OF INDEPENDENT ===> A
    VALUE OF DEPENDENT   ===> 000001

''' VALUE OF INDEPENDENT ===> B
    VALUE OF DEPENDENT   ===> 250001
```

*Figure 5-11   Variable dependency values*

The name of the independent variable should be set to reference the variable on which to base the values of MIN (in this case, TINST). The value pairs should then show which values of TINST cause equivalent values to be set for MIN.

The example shows MIN to be set to 000001 whenever TINST is A, and MIN to be set to 250001 whenever MIN is set to B. Further value pairs can be set (up to 360) for each variable in the table to handle corresponding values, more than enough to handle control data for parallel instances of a single job.

Having set up the dependent values for every variable that is needed to pass information into each instance, the running JCL then requires amendment to use the variable table.

Figure 5-12 shows the usage of an OPC TABLE directive, which is used to tie the job to the table containing the MIN and MAX variables. The value of the symbolic parameters can then be replaced with `&MIN` and `&MAX`. One requirement of dependent variables is that the independent variable must be referenced in the JCL at least once before the variables that depend on it. In this example, `&TINST` must appear in the JCL before `&MIN` and `&MAX`. Failure to do so results in the dependent variables returning their default values as opposed to their dependent values.

```
//*%OPC SCAN
//*%OPC TABLE NAME=DRWHOUSEKEEPING
//&OJOBNAME JOB (99999),'DR W HKP',USER=DAVISON,
//         CLASS=A,MSGCLASS=X
//*%OPC SETVAR TINST=SUBSTR(&OJOBNAME,8,1)
//*
//DRWHOUSE EXEC DRWHOUSE,TINST=&TINST,MIN=&MIN,MAX=&MAX
//
```

*Figure 5-12   Running JCL using dependent variables*

The JCL now can be identical for every instance, but returns unique JCL with the relevant set of values for each instance. So, in the case of 4-way parallelism, four members are created, each with identical content, but a member name matching the job name of each instance.

Figure 5-13 shows how this JCL is resolved for member DRWHOUPA.

```
//*>OPC SCAN
//*>OPC TABLE NAME=DRWHOUSEKEEPING
//DRWHOUPA JOB (99999),'DR W HKP',USER=DAVISON,
//         CLASS=A,MSGCLASS=X
//*>OPC SETVAR TINST=SUBSTR(DRWHOUPA,8,1)
//*
//DRWHOUSE EXEC DRWHOUSE,TINST=A,MIN=000001,MAX=250000
//
```

*Figure 5-13 Dependent variables that are resolved for DRWHOUPA*

In contrast, Figure 5-14 shows the same JCL being resolved differently when submitted under a different job name.

```
//*>OPC SCAN
//*>OPC TABLE NAME=DRWHOUSEKEEPING
//DRWHOUPB JOB (99999),'DR W HKP',USER=DAVISON,
//         CLASS=A,MSGCLASS=X
//*>OPC SETVAR TINST=SUBSTR(DRWHOUPB,8,1)
//*
//DRWHOUSE EXEC DRWHOUSE,TINST=B,MIN=250001,MAX=500000
//
```

*Figure 5-14 Dependent variables that are resolved for DRWHOUPB*

## Passing arguments to programs using PARMDD

If Tivoli Workload Scheduler for z/OS dependent variables are not an option, an alternative method to pass parameters to programs is the **PARMDD** argument of the **EXEC** statement. This argument allows the argument string to be passed inside a data set, rather than having to be coded directly within the JCL. This configuration then opens the possibility of creating individual PARM members for each job instance, which can be referenced through the TINST symbolic.

Table 5-1 shows the contents of a set of members that can be used to pass the parameters into the AUDIT step.

*Table 5-1 Members that are used to pass PARM values using PARMDD*

| Member name | Contents |
|---|---|
| AUDITA | DRWHOUPA 000001 250000 |
| AUDITB | DRWHOUPB 250001 500000 |
| AUDITC | DRWHOUPC 500001 750000 |
| AUDITD | DRWHOUPD 750001 999999 |

The AUDIT step can then be amended to use these members, as shown in Figure 5-15. Here PARMDD is used to point to the DD statement **AUDITPRM**. This statement references a data set containing the members that are outlined in Table 5-1 on page 42, and uses the TINST symbolic to point to the member name that is specific for this instance of the job.

```
//AUDIT    EXEC PGM=DRTRACK,
//         PARMDD=AUDITPRM
//STEPLIB  DD DISP=SHR,DSN=DR.LOADLIB
//TRACKLOG DD DISP=SHR,DSN=DR.VONWER.KSDS
//AUDITPRM DD DISP=SHR,DSN=DR.PARMLIB(AUDIT&TINST)
```

*Figure 5-15 Using PARMDD to access parameters through data sets*

## 5.2.5 Removing individual job members

Even though the techniques that are described so far reduced the JCL to a small number of identical lines for each instance of the job, the JCL still is repeated across multiple engines. Any changes to the underlying application processing must be repeated across multiple members.

There are two Tivoli Workload Scheduler for z/OS techniques that can be used to move the JCL to a single member. Other scheduling products might have similar functions:

- One uses the native OPC FETCH directive, which requires no special customization of Tivoli Web Services but does require a stub member to be created for each parallel instance. This stub member does not contain any application JCL, so although multiple members are required, the impact is only an initial one. Any subsequent changes to the application need to be made only to a single member.
- The other method uses the Job-Library-Read exit (EQQUX002), so an exit must be written and installed to make this work. However, this mechanism allows the operations in the application to refer directly to the single JCL skeleton. No instance level members are required, even if more parallel instances are introduced.

### Using the FETCH directive

To use the FETCH directive, the JCL should be stored with an instance neutral member name, which is effectively a job skeleton. For example, in the case of instances DRWHOUPA and DRWHOUPB, a skeleton member of DRWHOUP# can be used.

The individual jobs can then contain only two OPC directives, as shown in Figure 5-16. This simple JCL stub must exist for every job name instance into which the parallel job can be split. Then, at submission time, the contents of DRWHOUP# are fetched and resolved.

```
//*%OPC SCAN
//*%OPC FETCH MEMBER=DRWHOUP#
```

*Figure 5-16 A FETCH directive JCL stub*

Although this mechanism uses native features that do not require any customization of Tivoli Workload Scheduler for z/OS, there are some limitations that it imposes:

- The FETCH action cannot be nested. DRWHOUP# cannot contain any further FETCH directives.
- BEGIN and END OPC directives cannot be called from within a member that is called by FETCH. So, the skeleton itself cannot use any of the conditional INCLUDE and EXCLUDE capabilities that are available in the member that is called directly by Tivoli Workload Scheduler for z/OS.
- If the level of parallelism must be increased from the initial design, then additional JCL stub members must be created for each new job instance.

An alternative approach is creating a purely generic piece of stub JCL, such as the one shown in Figure 5-17.

```
//*%OPC SCAN
//*%OPC FETCH MEMBER=&SKELNAME
```

*Figure 5-17 Generic stub JCL*

This stub then uses a JCL variable table that is attached to the application occurrence, either by run cycles or at submission time, to set the name of the skeleton to use in each parallel instance. The same stub JCL can be used for every parallel instance of every job across the controller if a variable table is attached to the occurrence containing the variable SKELNAME.

This approach adds a level of flexibility. By using different variable tables on multiple run cycles within the same application, individual runs can use differing skeletons for the JCL. For example, the daily runs can call a different skeleton from the weekly runs of the same jobs.

### Using the Job-Library-Read exit (EQQUX002)

The Job-Library-Read exit is a documented exit point that is called just before the controller is about to fetch JCL that currently is not stored in the EQQJSnDS file. It has access to full information about the occurrence, the operation, the workstation, extended information, and user fields for the job being retrieved. Instead of simply fetching a member matching the job name from the EQQJBLIB, the exit can be driven by data in the scheduling object to select alternative member names, and DD statements, to fetch the JCL from. There is a sample available that can be used to fetch skeleton jobs by using the extended job name to point to the member to retrieve. To discover how to obtain this exit, see “Obtaining sample exits and the SOE toolkit” on page 108.

Installing and configuring the exit involves a few short steps. They are documented in the $INSTALL member of the library in which the exit is shipped. You should refer to this member for the latest instructions, but essentially they consist of the following steps:

1. Assemble and link-edit the exits using the JCL in members $ASMLX02, customizing the JCL for your installation. This step creates members EQQUX000 and EQQUX002.
2. Make sure that the exits are accessible to the controller either through the LNKLIST or in an authorized library in the controller STEPLIB.
3. Ensure that the **`EXITS`** statement in the controller configuration has **`CALL00(YES)`** and **`CALL02(YES)`** coded.
4. Create a member that is called USERPARM in the controller EQQPARM library to configure the exit. Figure 5-18 on page 45 gives the configuration that used for the examples that are shown in this publication.

5. Restart the controller.

```
GLOBLWTO=Y    Issue a WTO for each for each debug point - WEQGDBUG
DEBUGWTO=Y    Issue a WTO for each MODELSEL/MODELGEN error condition
MODSLWTO=Y    Issue an informational WTO for good MODELSEL GENed names
JCLINMSG=N    Insert EQQUXJ80I X cards added comment card into JCL
JCLPREFIX=USRJX
JCLDEFAULT=(UF,00,3)
###MODEL=NO
MODELSEL1=(XI,00,ALT=)
MODELGEN=(XI,04,8)
```

*Figure 5-18 A sample configuration for the exit*

The exit has two main driving points:

- Selection of an alternative library from which to retrieve the JCL.
- Selection of an alternative member from which to retrieve the JCL. If an alternative library is not selected, the alternative member name is ignored. To use skeleton jobs, they must all exist within a library other than the EQQJBLIB concatenation.

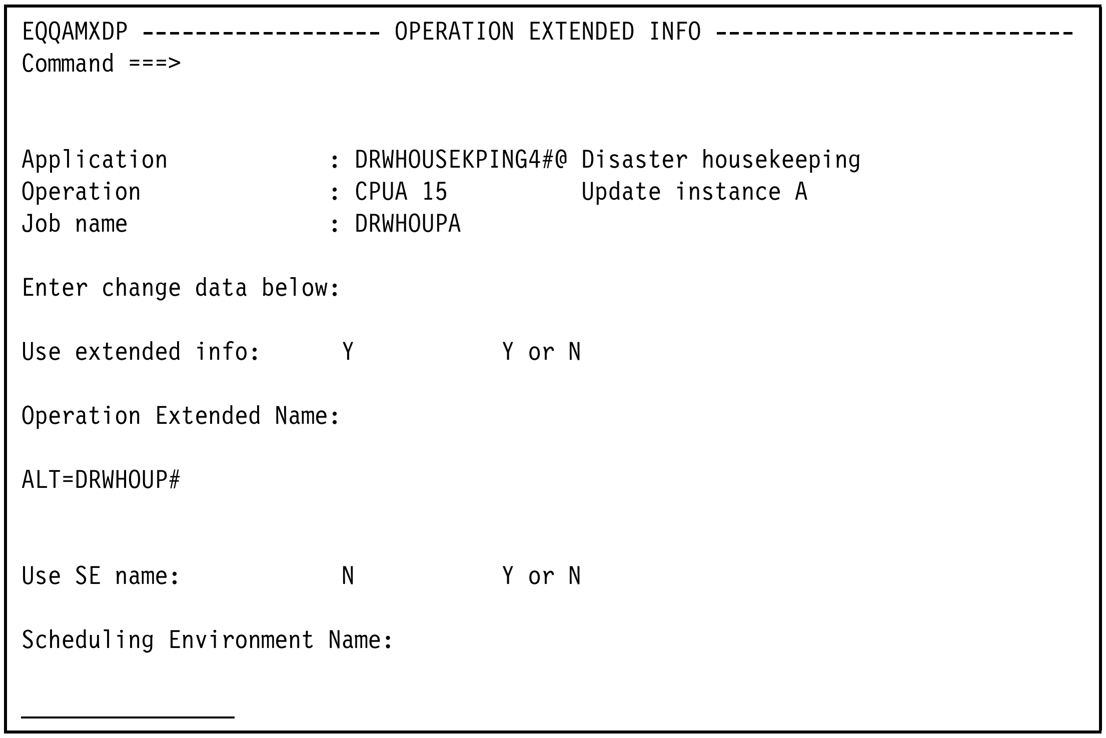
In the case of this configuration, the **JCLPREFIX** keyword instructs the exit to consider all DD statements beginning with USRJX as possible sources for JCL. When the controller is started with these exits installed, exit EQQUX000 finds and opens any DD statements in the controller JCL that begin with USRJX.

The **JCLDEFAULT** keyword then tells the exit to look for a user field (UF) with the same name as the value of **JCLPREFIX** (in this case, USRJX) and append to it the first three characters of the UF value (offset 00 for three characters). In the case of the examples for this publication, a DD statement of USRJXSKL was added to the controller. For any operations to use the exit, the member must be placed within the USRJXSKL library and a user field of USRJX must be specified with a value of SKL.

The **MODELSEL1** keyword tells the exit to look for a condition to tell it when to use an alternative member name. In this case, XI refers to the extended job name field, and the exit looks for the condition to start (offset 00) with ALT=. If it finds ALT= at the position, then MODELGEN tells it to look from offset 4 (immediately after the = sign) of the extended job name for eight characters to obtain the alternative member name. The ALT= tag that is specified in **MODELSEL1** is important because it reduces the risk of problems. If the extended job name is populated for another purpose, only operations where the extended job name begins with ALT= use an alternative member name. You can choose your own tag; it does not have to be ALT=. Choose something obvious and memorable that is not used in the extended job name field.

So, for example, coding user field USRJX=SKL and ALT=ZYGON in the extended job name field causes the exit to look for a member of ZYGON in library USRJXSKL. If the extended job name is coded without the user field, then the job is retrieved from EQQJBLIB with a member matching the job name.

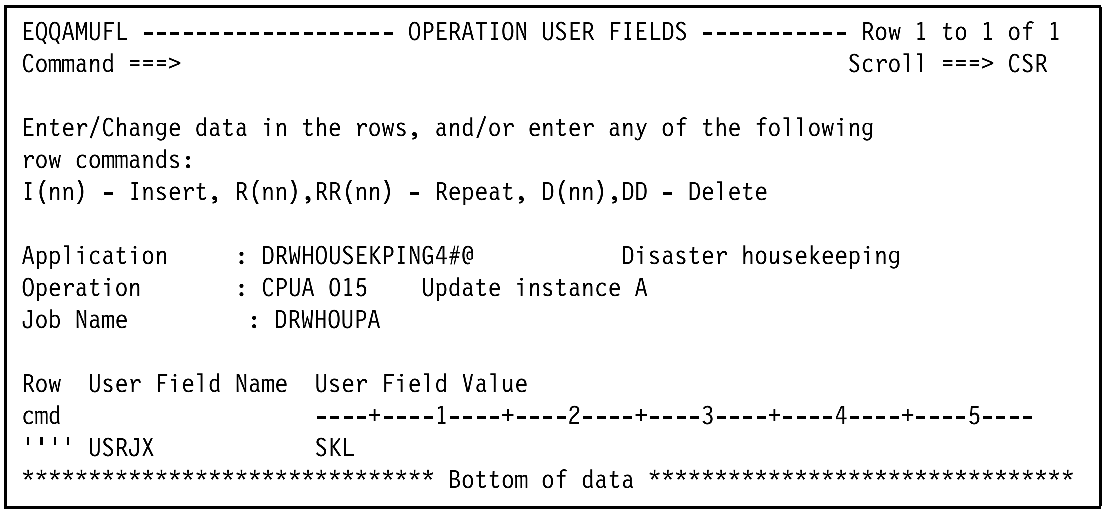
In the context of the example scenario, the job DRWHOUPA has an extended job name of ALT=DRWHOUP#, which is set from option 10 of the OPERATION DETAILS menu (EQQAMSP), as shown in Figure 5-19. This panel can also be reached by entering S.10 against the job in the OPERATIONS panel (EQQAMOSL or EQQAMOPL).



```
EQQAMXDP ------------------ OPERATION EXTENDED INFO --------------------------
Command ===>


Application           : DRWHOUSEKPING4#@ Disaster housekeeping
Operation             : CPUA 15          Update instance A
Job name              : DRWHOUPA

Enter change data below:

Use extended info:      Y           Y or N

Operation Extended Name:

ALT=DRWHOUP#


Use SE name:            N           Y or N

Scheduling Environment Name:

________________
```

Figure 5-19   Setting the extended job name to indicate the skeleton to use

To make the exit use this alternative member name, the user field USRJX must be set to point to the USRJXSKL library by setting a value of SKL. This value is set from option 12 of the OPERATION DETAILS menu (EQQAMSP), as shown in Figure 5-20. This panel can also be reached by entering S.12 against the job in the OPERATIONS panel (EQQAMOSL or EQQAMOPL).



```
EQQAMUFL ------------------- OPERATION USER FIELDS ----------- Row 1 to 1 of 1
Command ===>                                                 Scroll ===> CSR

Enter/Change data in the rows, and/or enter any of the following
row commands:
I(nn) - Insert, R(nn),RR(nn) - Repeat, D(nn),DD - Delete

Application     : DRWHOUSEKPING4#@          Disaster housekeeping
Operation       : CPUA 015    Update instance A
Job Name         : DRWHOUPA

Row  User Field Name  User Field Value
cmd                   ----+----1----+----2----+----3----+----4----+----5----
'''' USRJX            SKL
****************************** Bottom of data ********************************
```

Figure 5-20   Setting a user field to indicate that an alternative job source is required

To avoid having to navigate through many extra panels, complete the following steps:

1. Create one job instance from the OPERATIONS panel (EQQAMOSL/ EQQAMOPL).
2. Set the extended job name and user field.
3. Set the predecessors to make this job dependent on the start node.
4. Repeat this job for as many job instances that are needed and alter the job name accordingly.

The exit then inherits the extended job name and user fields, and the predecessor to the start node.

**Note:** Do not repeat a parallel instance job to create a non-parallel job. If you do not edit out the extended job name and user fields, the job attempts to use the skeleton job instead of the member matching the job name.

You can use the Job-Library-Read exit to use a single JCL skeleton for multiple jobs but with the following advantages over the FETCH method:

- JCL stub members are not needed. The pointer to the skeleton member is done directly from within the scheduling object.
- When the number of parallel instances is increased, no additional job members are needed.

## 5.2.6 Merging output data

If the parallel instances produce any output files, it is likely that this data must be collated in some manner during the end node. For fixed splits, where the names of all the output files are known, this is not a difficult task because the JCL can be written once. But where the content can vary from one run to the next, an automated solution is needed, as JCL alone cannot cope with this situation.

Generation data groups (GDG) cannot be used, even though JCL can merge an unknown number of entries. Only one job at once can write to a GDG, so it cannot be used as an output option for jobs that run in parallel.

### Generating merged JCL in the start node

When the processing in the start node is intelligent enough to know what output files will be created during that run, then one additional output of the start node should be a member containing a generated DD statement that includes all the files for that run, as shown in Figure 5-21.

```
//SORTIN   DD DISP=SHR,DSN=DR.WHOA.OUTFILE
//         DD DISP=SHR,DSN=DR.WHOB.OUTFILE
//         DD DISP=SHR,DSN=DR.WHOC.OUTFILE
//         DD DISP=SHR,DSN=DR.WHOD.OUTFILE
```

*Figure 5-21 Generated concatenation DD statement*

The generation process can write a member, for example $DD4SRT1, to a local project specific library rather than the main procedure libraries to cut down on risks of traffic and contention on the main procedure library. The local library can then be referenced by a **`JCLLIB`** statement to provide the source of the INCLUDE member.

Writing only the DD statement, as opposed to the whole job, allows the core of the JCL to remain fully visible in the operational environment. It is subject to normal searches, changes, and lifecycle management, leaving only the truly dynamic part contained within a program.

The generated member can then be used later in the end node as part of the merge process by using a JCL include, as shown in Figure 5-22.

```
//         JCLLIB ORDER='DR.JCL.INCLIB'
//*
//STEP010  EXEC PGM=SORT
//SORTIN   INCLUDE MEMBER=$DD4SRT1
//SORTOUT  DD DSN=DR.WHO.SORTED,
//            DISP=(NEW,CATLG),DSORG=PS,
//            SPACE=(TRK,(10,10)),RECFM=FB,LRECL=101
//SYSOUT   DD SYSOUT=*
//SYSIN    DD *
  SORT FIELDS=(51,4,CH,A,1,50,CH,A,64,3,PD,D)
/*
```

*Figure 5-22 Merged data sets referenced by INCLUDE*

### Generating JCL in the end node

An alternative to generating JCL in the start node is to use some generic automation to generate it in the end node.

To accomplish this task, you need the following elements in place:

- A reliable naming convention so that a data set prefix or mask can be specified that only returns the data sets you want to merge.
- A disciplined data set lifecycle so that input data sets are deleted or renamed after they are merged. Having this element avoids leakage of data from previous runs.

It should then be possible to create an automated process to find the matching data sets and produce a JCL INCLUDE member.

During the research for this publication a generic REXX utility was written that performs this function. You can find this utility in “REXX utility JCLCATDD” on page 108.

## 5.3 Scheduling techniques

After the JCL is built to form the start node, parallel instance, and end node processing, you must consider scheduling. The basic structure always is based upon the flow that is outlined in Figure 5-1 on page 30, but the techniques that can be used to achieve this are numerous.

### 5.3.1 Static versus dynamic scheduling

The key factor to deciding which technique to use is whether the scheduling is going to be static or dynamic. In static scheduling, the same jobs run every time that this process is scheduled. In dynamic scheduling, decisions can be made to add or remove jobs each time it runs.

From an operational perspective, dynamic scheduling has an advantage of removing unnecessary job submission from processing. Although this might not seem important when running a simple process that might (or might not) need four jobs running in parallel, when scaled up to hundreds or thousands of jobs that are split into parallel instances, the picture is different. This situation can lead to a large amount of unwanted JCL conversion time and initiator usage alone, regardless of how much resource each unnecessary job then goes on to consume.

However, the ability to perform dynamic scheduling relies entirely on the design of the application. If the application cannot decide about what workload is needed from day to day, then static scheduling is the only choice. So, although dynamic scheduling can have operational benefits, it might be difficult to achieve.

If an application can be engineered in a way to be dynamic, there are two design options to consider: conditional execution and dynamic splitting.

#### Conditional execution

This is where the workload is split into instances in a manner that does not vary from run to run, but it might be possible to run a process in the start node to determine that on a particular day one or more of those instances might have no data to process.

For example, a business might split its workload into instances that are based upon geographic location, each of which might have different public holidays. In this kind of application, avoiding the submission of jobs on days when the location has a public holiday makes for prioritized usage of resources.

Another example is to query the input data for that day to see whether there are records that apply to each individual instance, and not run any instances with no data to process that day.

#### Dynamic data splitting

This is where the application can redefine the split of the data between parallel instances every time the process is run. This might result in static scheduling, with the same number of instances being run each time that it runs, making the overall elapsed time vary, but keeping it evenly distributed across instances. Alternatively, it might result in dynamic scheduling, varying the number of instances in parallel while attempting to keep the overall elapsed time within certain limits.

This approach is not compatible with scenarios where the split is driven by database partition ranges, as to do so requires redefining partitions for each run, which probably negates any time savings that are made by rebalancing the split each run.

### 5.3.2 Static parallel scheduling techniques

In most cases of parallel scheduling, the number of jobs that are involved is small. Typically, with less than 10 jobs in total in the start and end nodes, and even with a 32-way split, the overall number is not large. Most static parallel scheduling comfortably fits within a single scheduling object.

## Single scheduling object

Figure 5-23 shows a simple static scheduling object design. In this case, application definition DRWHOUSEKPING4#@ has eight operations, each represented by a blue box. Each box shows the workstation and operation number on the first line, and the job name on the second line. In Tivoli Workload Scheduler for z/OS, the workstation name defines what type of job is running, and the operation number is a unique key. The example workstation, NONR, defines dummy operations that do nothing; they are marked complete after all dependencies are met. The example workstation, CPUA, defines batch jobs.

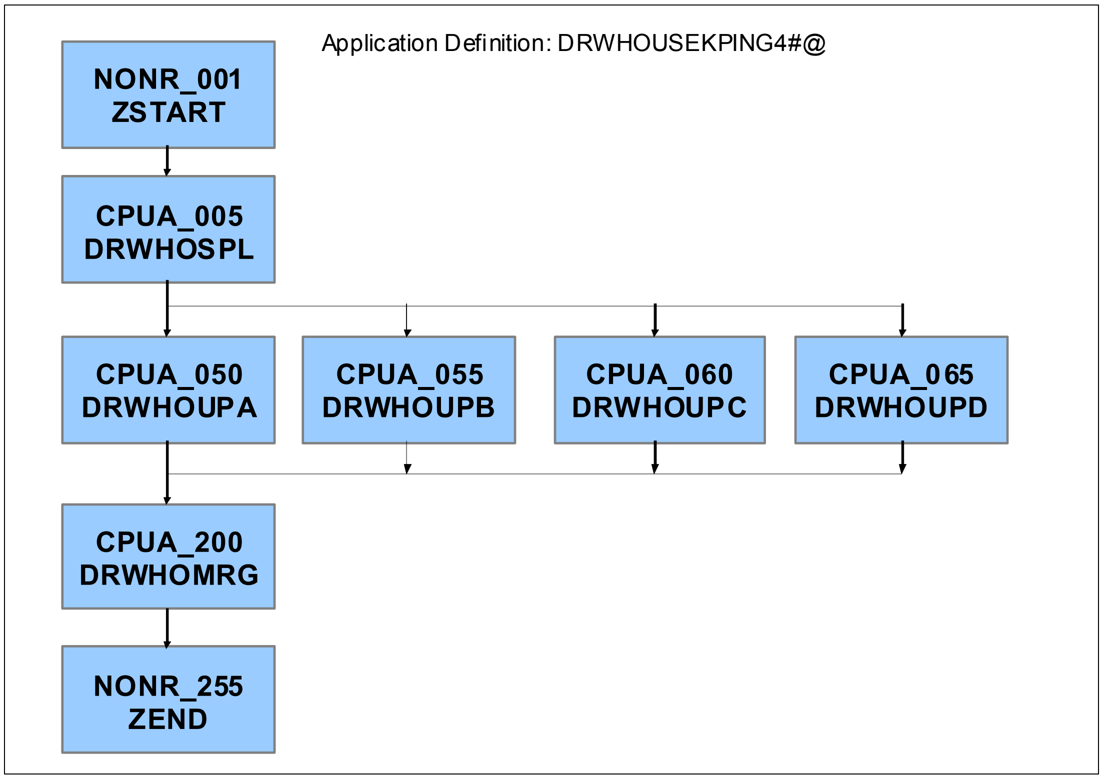


*Figure 5-23 Static application definition*

The scheduling object begins and ends with a dummy operation. The reason behind this practice is that it encapsulates the internal dependency structure, protecting external dependencies from changes to the internal content. For example, supposed that job ZSTART holds all the external predecessors to this process, and ZEND is the point to which all external successors are defined. Then, if a new job is added after DRWHOMRG, or DRWHOSPL is deleted, if internal dependencies are maintained to ZSTART and ZEND, no external references must be changed.

The operation numbers for DRWHOUPA and DRWHOMERG show a distinct gap from their predecessors. This helps show the structure in the form of start node, instances, and end node. Consider creating a site guideline to define the starting operation number of the instances and end node to make it clear which operations belong to each part of the standard structure.

### Multiple scheduling objects

In the simple examples that are presented so far, a single job was considered for splitting. On a larger scale, it is possible to split many jobs across an entire business application by using an economy of scale combining all of the start node and end node activities, and then running many instances of multi-job application batch in parallel. In these cases, instead of combining the whole process into a single scheduling object, a more flexible approach is to separate the start node, end node, and parallel processing instances into separate scheduling objects.

Figure 5-24 shows that DRWHOUSEKPING4## contains the start node processing, DRWHOUSEKPING4#9 contains the end node processing, and DRWHOUSEKPING4#A to DRWHOUSEKPING4#D handle the multiple jobs for each instance.

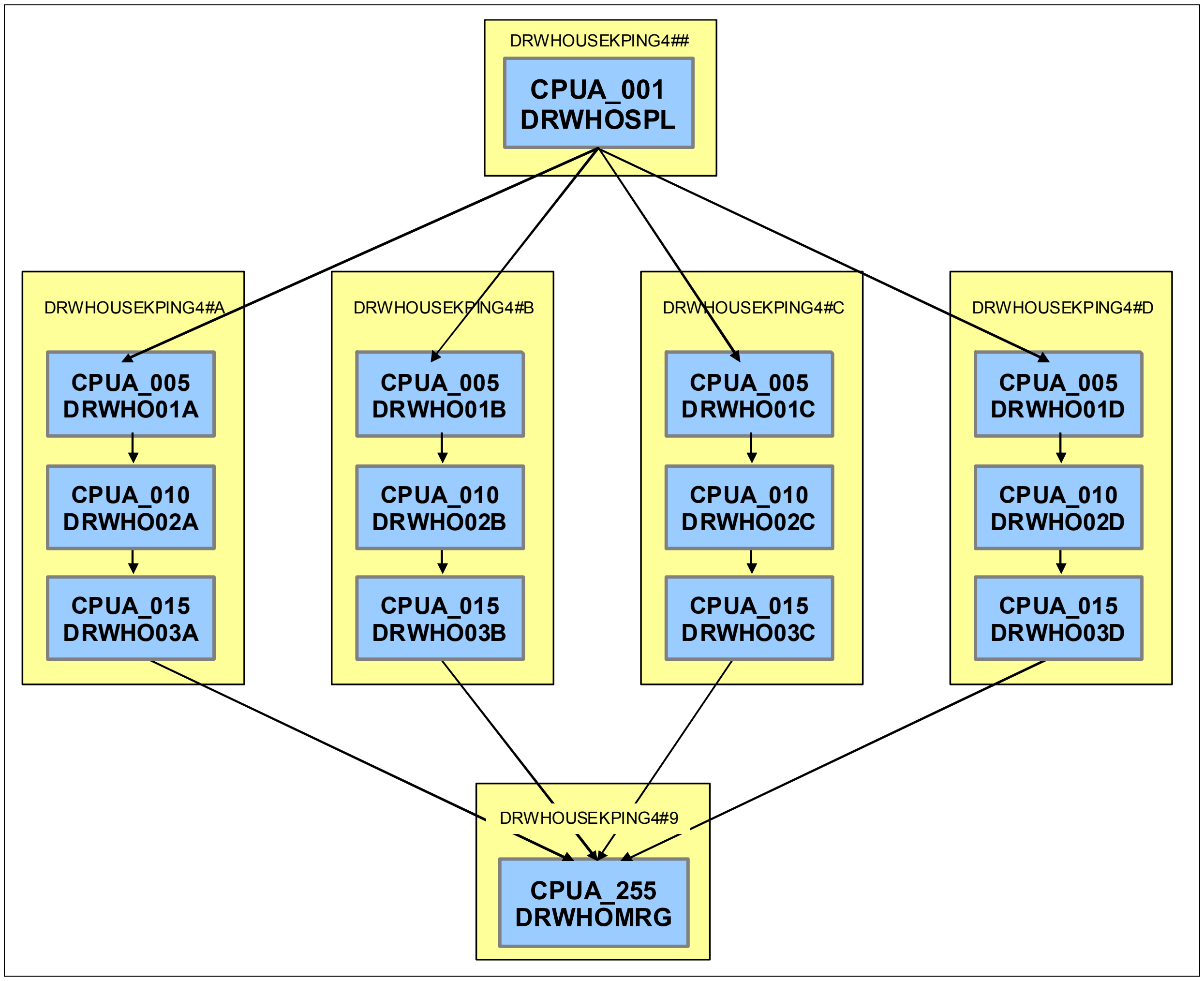


*Figure 5-24 Multi-job split*

## 5.3.3 Dynamic parallel scheduling techniques

Although the ability to drive dynamic parallel scheduling relies entirely on the design of the application, there are several scheduling techniques that can be used if the application design permits.

### Conditional execution

This is where the scope of the split is the same every run, but there might not always be data for each instance in every run. The aim of conditional execution is to not introduce a job to JES if it has nothing to do, which removes unnecessary consumption of resources.

Many workload automation products have features by which jobs can be run conditionally, usually by interrogating return code of previous jobs. In the case of Tivoli Workload Scheduler for z/OS, this feature is known as *conditional dependencies*. Using this feature, conditional dependencies can be made between jobs. These dependencies can react to the status of a job, the return code, or a range of return codes. Conditions can be complex and include dependencies to many different jobs and steps. Tivoli Workload Scheduler for z/OS waits until all the prerequisite jobs and steps end before deciding whether to run the job or not. If all the conditions are not met, then the job is skipped and set to a status X to indicate that it is excluded from that run.

In this example, we have a business that splits its account processing by the senior partner that is responsible for the customer. If there are no transactions for a particular partner, then there is no need to run batch for that instance.

Section 3.2.2, “Checking data” on page 19 gives examples about how to issue return codes that are based on whether there is data that is contained within files, so one option to control conditional execution is to have an individual step for each instance input file and use conditional dependencies that are targeted at the appropriate step.

Figure 5-25 on page 53 shows an example job that checks whether there is data in each of the four partners’ data files. There is no IF-THEN-ELSE construct required here because we want RC=0000 returned if data is found, and a nonzero return code if no data is found. The conditional dependencies for each partner can be targeted at the step corresponding to their data file.

```
//LEGALSPL JOB CLASS=A,MSGCLASS=X,USER=DELGADO
//* DATA CHECK FOR ECCLESTON'S ACCOUNTS
//CHECK#CE EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//CHECKME  DD DISP=SHR,DSN=DEAN.LEGAL.CE.DATA
//DUMPME   DD SYSOUT=*
//SYSIN    DD *
 REPRO INFILE(CHECKME) OUTFILE(DUMPME) COUNT(1)
/*
//* DATA CHECK FOR TENNANT'S ACCOUNTS
//CHECK#DT EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//CHECKME  DD DISP=SHR,DSN=DEAN.LEGAL.DT.DATA
//DUMPME   DD SYSOUT=*
//SYSIN    DD *
 REPRO INFILE(CHECKME) OUTFILE(DUMPME) COUNT(1)
/*
//* DATA CHECK FOR SMITH'S ACCOUNTS
//CHECK#MS EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//CHECKME  DD DISP=SHR,DSN=DEAN.LEGAL.MS.DATA
//DUMPME   DD SYSOUT=*
//SYSIN    DD *
 REPRO INFILE(CHECKME) OUTFILE(DUMPME) COUNT(1)
/*
//* DATA CHECK FOR CAPALDI'S ACCOUNTS
//CHECK#PC EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//CHECKME  DD DISP=SHR,DSN=DEAN.LEGAL.PC.DATA
//DUMPME   DD SYSOUT=*
//SYSIN    DD *
 REPRO INFILE(CHECKME) OUTFILE(DUMPME) COUNT(1)
/*
```

*Figure 5-25 Data check job*

**Note:** If you want to use step-level conditional dependencies, then you must either set **`EWTROPTS STEPEVENTS(ALL)`** or use the **`SDEPFILTER`** keyword in the tracker configuration. These settings ensure that all the step events are passed through to the controller for jobs that are subject to conditional dependencies. For more information, see *Workload Scheduler for z/OS Customization and Tuning*, SC32-1265.

To use this job, create an application definition that is similar to the diagram that is shown in Figure 5-26. Here are some Important things to note about this design:

- The black lines are normal dependencies and the red lines are conditional dependencies.
- Conditional dependencies are not considered when checking the consistency of an application. All operations must be connected to each other, directly or indirectly, by normal dependencies. Tivoli Workload Scheduler for z/OS does not allow an application definition to be saved that does not meet this criteria.
- The normal dependency between operation 005 and operation 255 is important, as Tivoli Workload Scheduler for z/OS fails if it finds no valid onward route. So, if all four files are empty, causing all of the conditional dependencies to be false, then an error occurs if there is no other valid route from operation 5 to another operation in the application definition.

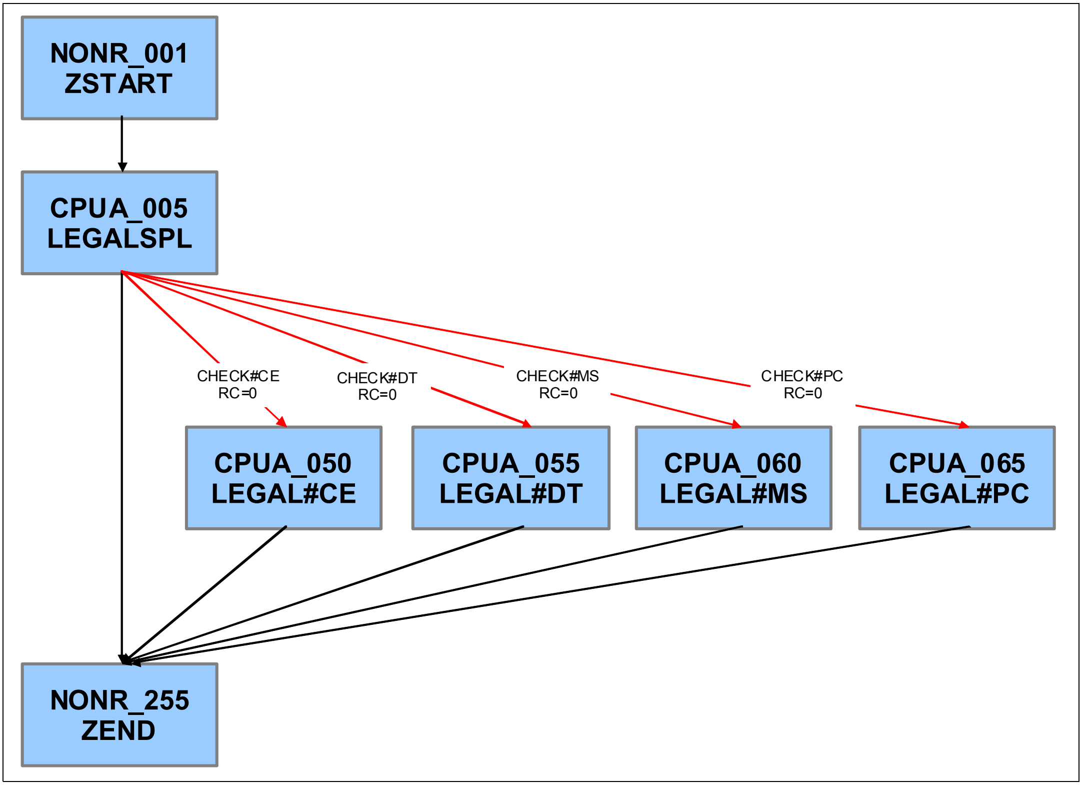


*Figure 5-26 Step-level conditional dependencies*

To create a conditional dependency, enter S.1 against the job in the OPERATIONS panel (EQQAMOSL or EQQAMOPL) and then enter COND. Figure 5-27 on page 55 shows the conditional dependency being made to step CHECK#CE of job LEGALSPL.

```
EQQAMCCP ------------- CONDITION DEPENDENCIES DEFINITION ----- Row 1 to 1 of 1
Command ===>                                                Scroll ===> CSR

To define a condition dependency enter/change data in the rows, using any
of the following row commands:
I(nn) - Insert, R(nn),RR(nn) - Repeat, D(nn),DD - Delete, T - Dependency
resolution criteria

Application     : LEGALDAILYACC#@@           Daily legal accounting
Operation       : CPUA 050    LEGAL#CE       Eccleston

Rule:
  Specify the number of condition dependencies that need to be verified
  to make the condition true 000 . Leave 0 for all of them.

Row Oper     Application Id   Jobname  StepName ProcStep Co Co St  Ret.Code
cmd ws.  no. (ext Adid only)                             Ty OP Val Val1 Val2
''' CPUA 005 ________________ LEGALSPL ________ CHECK#CE RC EQ _   0000 ____
****************************** Bottom of data *******************************
```

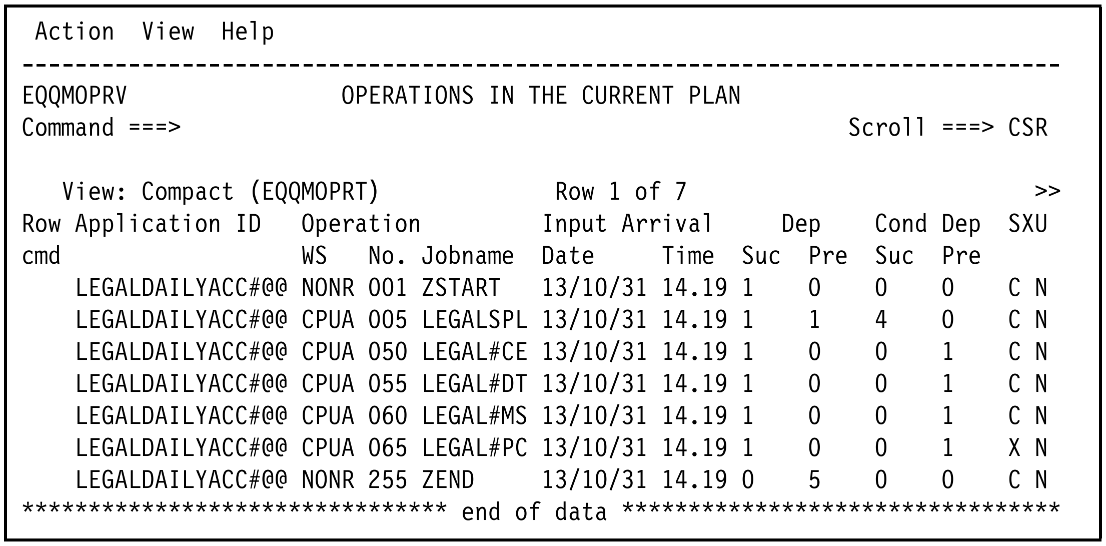
Figure 5-27   Defining a conditional dependency

When the application definition is run, the first job LEGALSPL runs to check each file. In Figure 5-28, you can see CHECK#PC ended with return code 4 to indicate no data.

```
-JOBNAME  STEPNAME PROCSTEP    RC   EXCP    CPU    SRB  CLOCK   SERV  PG   PAGE
-LEGALSPL          CHECK#CE    00     49    .00    .00    .00    284   0      0
-LEGALSPL          CHECK#DT    00     55    .00    .00    .00    270   0      0
-LEGALSPL          CHECK#MS    00     45    .00    .00    .00    186   0      0
-LEGALSPL          CHECK#PC    04     48    .00    .00    .00    263   0      0
-LEGALSPL ENDED.  NAME-                     TOTAL CPU TIME=   .00  TOTAL ELAPSE
$HASP395 LEGALSPL ENDED
```

Figure 5-28   Check job return codes

This then results in three of the instances running, but job LEGAL#PC is skipped, as shown by the status of X against the job in Figure 5-29.



```
 Action  View  Help
------------------------------------------------------------------------------
EQQMOPRV               OPERATIONS IN THE CURRENT PLAN
Command ===>                                                  Scroll ===> CSR

   View: Compact (EQQMOPRT)            Row 1 of 7                          >>
Row Application ID   Operation         Input Arrival     Dep    Cond Dep  SXU
cmd                  WS   No. Jobname  Date     Time  Suc  Pre  Suc  Pre
    LEGALDAILYACC#@@ NONR 001 ZSTART   13/10/31 14.19 1    0    0    0    C N
    LEGALDAILYACC#@@ CPUA 005 LEGALSPL 13/10/31 14.19 1    1    4    0    C N
    LEGALDAILYACC#@@ CPUA 050 LEGAL#CE 13/10/31 14.19 1    0    0    1    C N
    LEGALDAILYACC#@@ CPUA 055 LEGAL#DT 13/10/31 14.19 1    0    0    1    C N
    LEGALDAILYACC#@@ CPUA 060 LEGAL#MS 13/10/31 14.19 1    0    0    1    C N
    LEGALDAILYACC#@@ CPUA 065 LEGAL#PC 13/10/31 14.19 1    0    0    1    X N
    LEGALDAILYACC#@@ NONR 255 ZEND     13/10/31 14.19 0    5    0    0    C N
******************************** end of data *********************************
```

*Figure 5-29 Conditional dependencies in action*

## Dynamic split

A common dynamic split scenario involves adding more instances as data increases, with the split process evenly distributing the data between the instances. So, for example, for up to 100000 records, only one instance must run. For 100001 - 200000 records, two instances must run. For 200001 - 300000 records, three instances must run. For 300001 records and above, four instances must run.

This scenario requires the creation of an application program to determine the number of input records for the run and set a return code 0 - 4. Zero means that no input records are allowed in a run.

Then, another form of conditional dependency can be used. Instead of just checking whether a return code is set, it uses relational checks to see whether the return code is greater than or equal to the value from which it needs to trigger.

Figure 5-30 on page 57 shows the basic design of a dynamic split. Here, the return code check rises from left to right, so LEGALCNA runs for return codes 1, 2, 3, and 4, and LEGALCNB runs only for 2, 3, and 4. The trend continues for as many instances as the process is designed to handle. Notice the dependency between operations 005 and 255 to handle the scenario where no records are found.

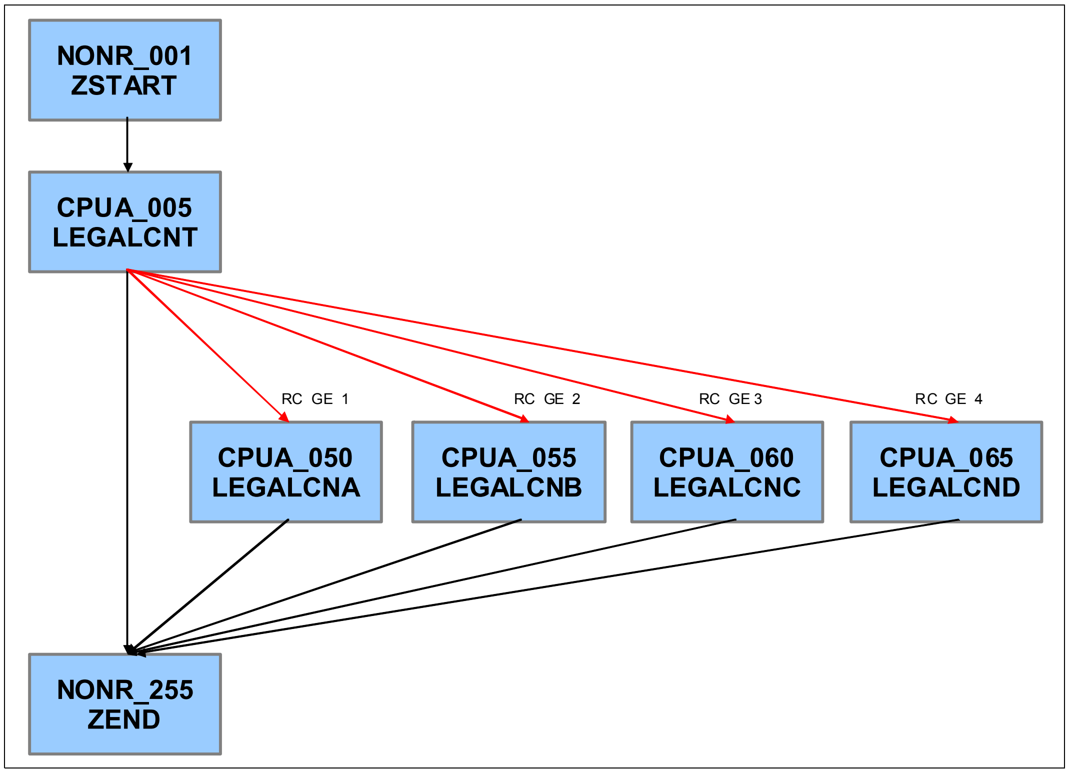
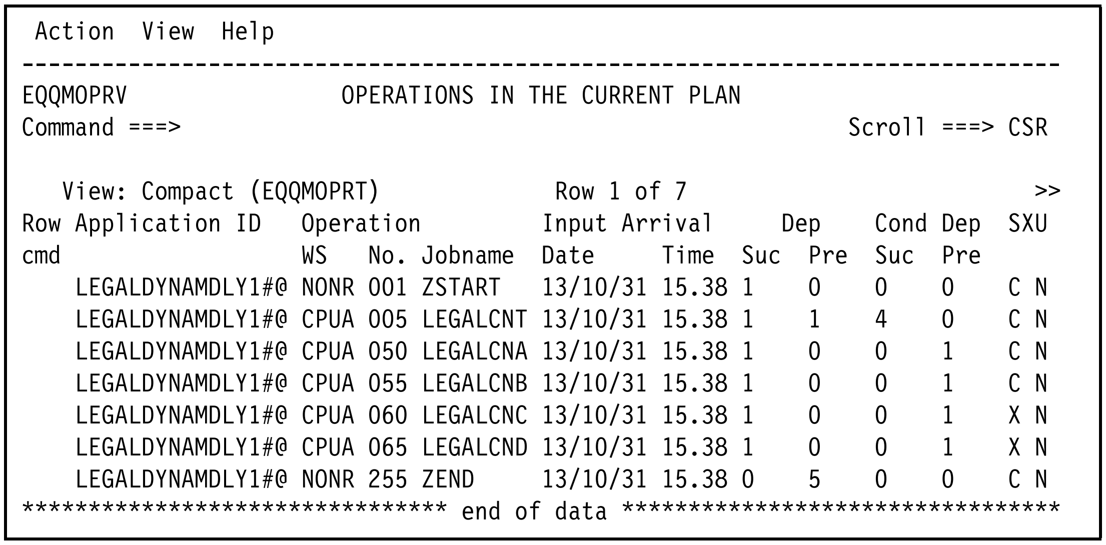


*Figure 5-30 Dynamic split design*

So, if LEGALCNT runs and ends with return code 2, both LEGALCNA and LEGALCNB run, with LEGALCNC and LEGALCND being skipped and set to status X, as shown in Figure 5-31.



```
  Action  View  Help
------------------------------------------------------------------------------
EQQMOPRV                OPERATIONS IN THE CURRENT PLAN
Command ===>                                                  Scroll ===> CSR

   View: Compact (EQQMOPRT)              Row 1 of 7                         >>
Row Application ID   Operation          Input Arrival      Dep     Cond Dep   SXU
cmd                  WS   No. Jobname   Date     Time  Suc  Pre    Suc  Pre
    LEGALDYNAMDLY1#@ NONR 001 ZSTART    13/10/31 15.38 1    0      0    0     C N
    LEGALDYNAMDLY1#@ CPUA 005 LEGALCNT  13/10/31 15.38 1    1      4    0     C N
    LEGALDYNAMDLY1#@ CPUA 050 LEGALCNA  13/10/31 15.38 1    0      0    1     C N
    LEGALDYNAMDLY1#@ CPUA 055 LEGALCNB  13/10/31 15.38 1    0      0    1     C N
    LEGALDYNAMDLY1#@ CPUA 060 LEGALCNC  13/10/31 15.38 1    0      0    1     X N
    LEGALDYNAMDLY1#@ CPUA 065 LEGALCND  13/10/31 15.38 1    0      0    1     X N
    LEGALDYNAMDLY1#@ NONR 255 ZEND      13/10/31 15.38 0    5      0    0     C N
******************************** end of data *********************************
```

*Figure 5-31 Dynamic split in action*

## Direct insertion

Another dynamic technique is not to schedule the instances but add them on demand for each run.

So, the start node processing must include a job that decides which instances are needed for that run and add them to the workload. Figure 5-32 shows the flow of dependencies for such a scenario.

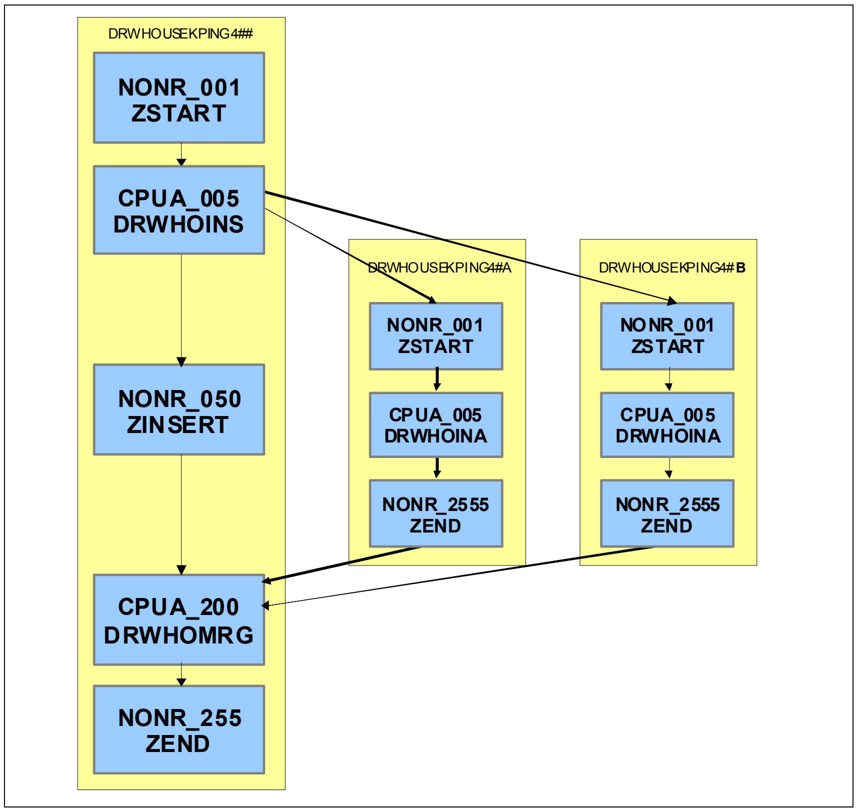


*Figure 5-32 Direct insertion diagram*

The insertion job can either decide how many instances to insert, and then conditionally run insertion steps for each instance, or decide how many instances to insert and then create the appropriate control statements for the INSERT step.

Adding extra workload on demand can, in certain circumstances, result in dependencies not being honored in the same way they would if the same scheduling objects were planned. So, it is safer to add explicitly the dependencies at insertion time to be sure all the connections are made.

For Tivoli Workload Scheduler for z/OS, the Program Interface (PIF) permits this kind of action to be performed. The Batch Command Utility automatically commits each individual command to the Current Plan as they are issued, so is not always suitable for a sequence of commands where you want all of the instructions to be obeyed before anything starts running. For this example, the SOE toolkit is used to show how it can be done, as it does not commit the commands until they all are applied to the current plan. Details about how to obtain the SOE toolkit can be found in “Obtaining sample exits and the SOE toolkit” on page 108.

Figure 5-33 on page 59 shows an example that inserts two new instances in to the current plan by using the SOE toolkit.

```
//CALLSOE  EXEC EQQYXJCL,
//         SUBSYS=T91C
//SYSIN    DD *
VARSUB SCAN
INSERT CPOC ADID(DRWHOUSEKPING4#A) IA(!OYMD1!OHHMM)
MODIFY CPOP OPNO(001)
INSERT CPPRE PREADID(!OADID) PREIA(!OYMD1!OHHMM) PREOPNO(5)
MODIFY CPOP OPNO(255)
INSERT CPSUC SUCADID(!OADID) SUCIA(!OYMD1!OHHMM) SUCOPNO(200)
INSERT CPOC ADID(DRWHOUSEKPING4#B) IA(!OYMD1!OHHMM)
MODIFY CPOP OPNO(001)
INSERT CPPRE PREADID(!OADID) PREIA(!OYMD1!OHHMM) PREOPNO(5)
MODIFY CPOP OPNO(255)
INSERT CPSUC SUCADID(!OADID) SUCIA(!OYMD1!OHHMM) SUCOPNO(200)
```

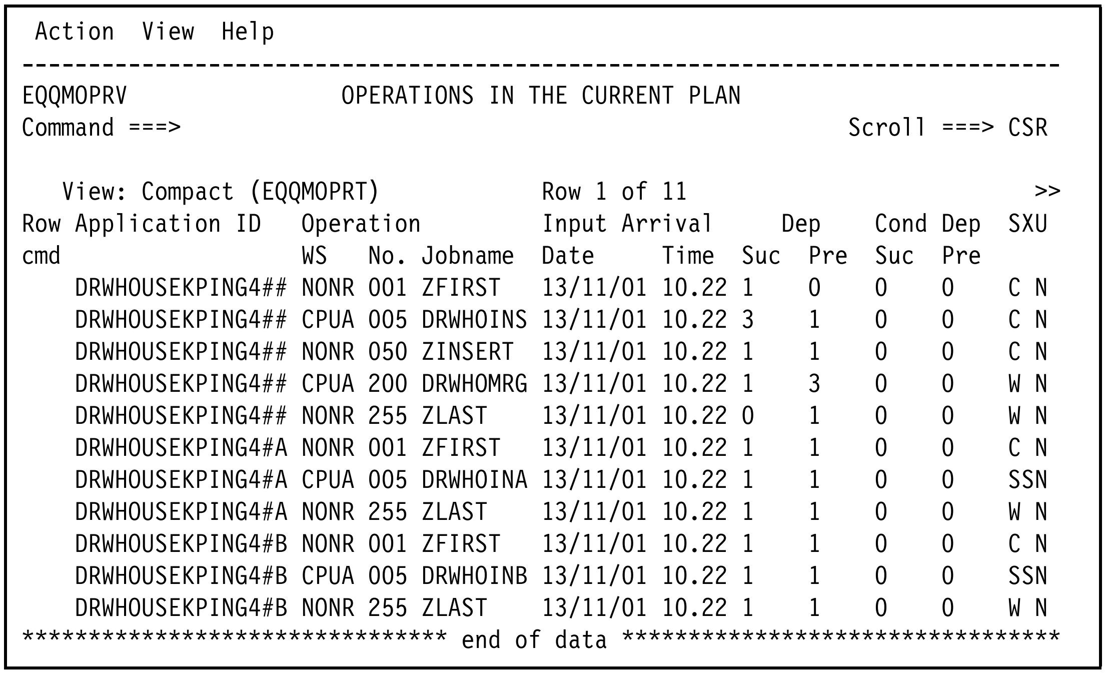
*Figure 5-33 Example of direct insertion using SOE*

The command syntax of SOE is easily translated in to Tivoli Workload Scheduler for z/OS PIF, with the first two words of each statement representing the action code and resource code, with the rest of the statement being argument name and value pairs. Without SOE, the same process can be performed by writing your own program to call the PIF.

Here is a step-by-step explanation of what each statement is doing:

1. **VARSUB SCAN** tells SOE to locate the job that it is running under within the current plan, so that it can obtain information about the occurrence controlling the job. This information then populates SOE variables, which can be referenced in other commands. In SOE terms, the variables begin with "!" as opposed to "&" for Tivoli Workload Scheduler for z/OS variables, but use equivalent names when appropriate. Without SOE, the Tivoli Workload Scheduler for z/OS variables &OYMD1, &OHHMM, and &OADID can be passed into the PIF program from the JCL and provide the same information into the process in place of the SOE variables.
2. **INSERT CPOC** adds a current plan occurrence for DRWHOUSEKPING4#A, using the same input arrival date and time as the controlling occurrence DRWHOUSEKPING4## to make the connection between the two clear.
3. **MODIFY CPOP** selects operation 001 in the newly inserted occurrence in preparation for adding a predecessor to it.
4. **INSERT CPPRE** adds a predecessor to operation 001 to make it wait for operation 005 in DRWHOUSEKPING4## (!OADID resolves to the occurrence running this process).
5. The second **MODIFY CPOP** selects operation 255 in the newly inserted occurrence in preparation for adding a successor to it.
6. **INSERT CPSUC** adds a successor operation to operation 255 to make operation 200 in DRWHOUSEKPING4## wait for this occurrence to complete.
7. The remaining statements repeat the same process for a second insertion, in this case for DRWHOUSEKPING4#B.

Then, unseen parts of the command statements SOE automatically perform an **EXECUTE MCPBLK** to commit all of these actions to the current plan. Also unseen in the command statements is when SOE automatically performs an **INIT** action at the beginning of the program and a **TERM** at the end, as shown in Figure 5-34.



```
 Action  View  Help
------------------------------------------------------------------------------
EQQMOPRV                OPERATIONS IN THE CURRENT PLAN
Command ===>                                                  Scroll ===> CSR

   View: Compact (EQQMOPRT)           Row 1 of 11                            >>
Row Application ID   Operation         Input Arrival     Dep    Cond Dep  SXU
cmd                  WS   No. Jobname  Date     Time  Suc  Pre  Suc  Pre
    DRWHOUSEKPING4## NONR 001 ZFIRST   13/11/01 10.22 1    0    0    0    C N
    DRWHOUSEKPING4## CPUA 005 DRWHOINS 13/11/01 10.22 3    1    0    0    C N
    DRWHOUSEKPING4## NONR 050 ZINSERT  13/11/01 10.22 1    1    0    0    C N
    DRWHOUSEKPING4## CPUA 200 DRWHOMRG 13/11/01 10.22 1    3    0    0    W N
    DRWHOUSEKPING4## NONR 255 ZLAST    13/11/01 10.22 0    1    0    0    W N
    DRWHOUSEKPING4#A NONR 001 ZFIRST   13/11/01 10.22 1    1    0    0    C N
    DRWHOUSEKPING4#A CPUA 005 DRWHOINA 13/11/01 10.22 1    1    0    0    SSN
    DRWHOUSEKPING4#A NONR 255 ZLAST    13/11/01 10.22 1    1    0    0    W N
    DRWHOUSEKPING4#B NONR 001 ZFIRST   13/11/01 10.22 1    1    0    0    C N
    DRWHOUSEKPING4#B CPUA 005 DRWHOINB 13/11/01 10.22 1    1    0    0    SSN
    DRWHOUSEKPING4#B NONR 255 ZLAST    13/11/01 10.22 1    1    0    0    W N
******************************** end of data ********************************
```

*Figure 5-34 Dynamic insertion in action*

Figure 5-34 shows the two inserted occurrences DRWHOUSEKPING4#A and DRWHOUSEKPING4#B, with their input arrival matched to DRWHOUSEKPING4## and job DRWHOMRG, which has three predecessors.

### 5.3.4 Initialization contention

Starting many almost identical jobs in parallel might lead to contention as the process initializes itself. Early processes within the application program might hit common resources as they initialize, but later in the process they access split resources.

If the contention cannot be removed from the program, one method to deal with the contention is to introduce progressively increasing short delays at the head of each instance.

So, for example, the first 10 seconds of a program access a common resource to record information about the process about to be run. If two jobs start at the same time, there is a risk one of them will fail. Once past the first 10 seconds, the risk of contention is much lower.

In Tivoli Workload Scheduler for z/OS terms, there is a type of workstation that is known as a WAIT workstation. All this workstation does is pause the workflow for a specified time before continuing with the next job.

Figure 5-35 on page 61 shows a WAIT workstation that staggers each of the parallel operation start points by 10 seconds. From a technical standpoint, Tivoli Workload Scheduler for z/OS cannot wait for zero seconds, so the first instance most likely does not have a wait operation, or the first instance uses a non-reporting (dummy) workstation in place of the wait workstation to keep the structure balanced across all the instances.

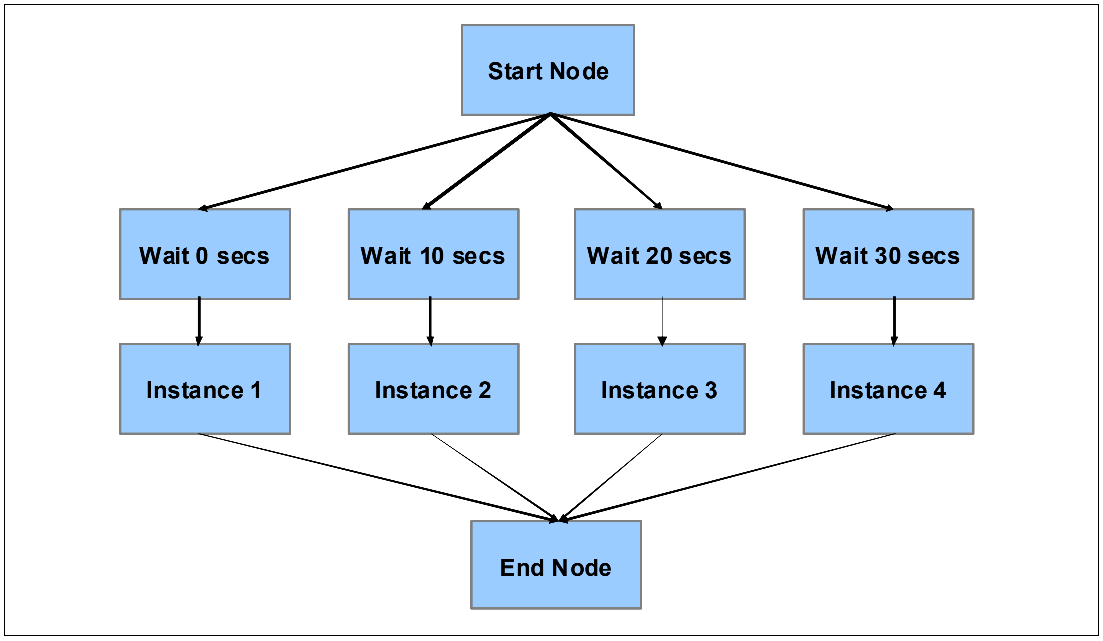


*Figure 5-35 Staggered start point*

### 5.3.5 Minimizing scheduling maintenance

In 5.2.3, “Minimizing JCL maintenance” on page 38, techniques were examined to reduce the amount of maintenance that is required across multiple instances if changes to the underlying application function require JCL changes to be implemented. Similarly, changes in scheduling requirements can result in maintenance being repeated across any scheduling elements that are duplicated for parallel instances.

There are methods that can be employed to reduce some of the impact such changes might have.

#### Externalizing planning rules

In cases where the start node, end node, and instances are in separate scheduling objects, and each of these objects are always planned upon the same day, then separating the planning rules from the scheduling objects themselves potentially reduces maintenance.

Even in cases where most of the scheduling objects run on the same days, but there are minor exceptions for some of the objects, some workload automation tools allow the core planning rules to be included in an external object, with variations handled as overrides in the scheduling object.

Taking this approach means that whenever planning rules must be changed, they need to be applied only to one external object instead of being applied to every individual scheduling object.

In Tivoli Workload Scheduler for z/OS terms, there is an object that is called a *runcycle group* that allows the planning rules to be externalized. Figure 5-36 shows a sample of a disaster recovery runcycle group for housekeeping, where the planning rules define a `2pm` input arrival every Saturday.

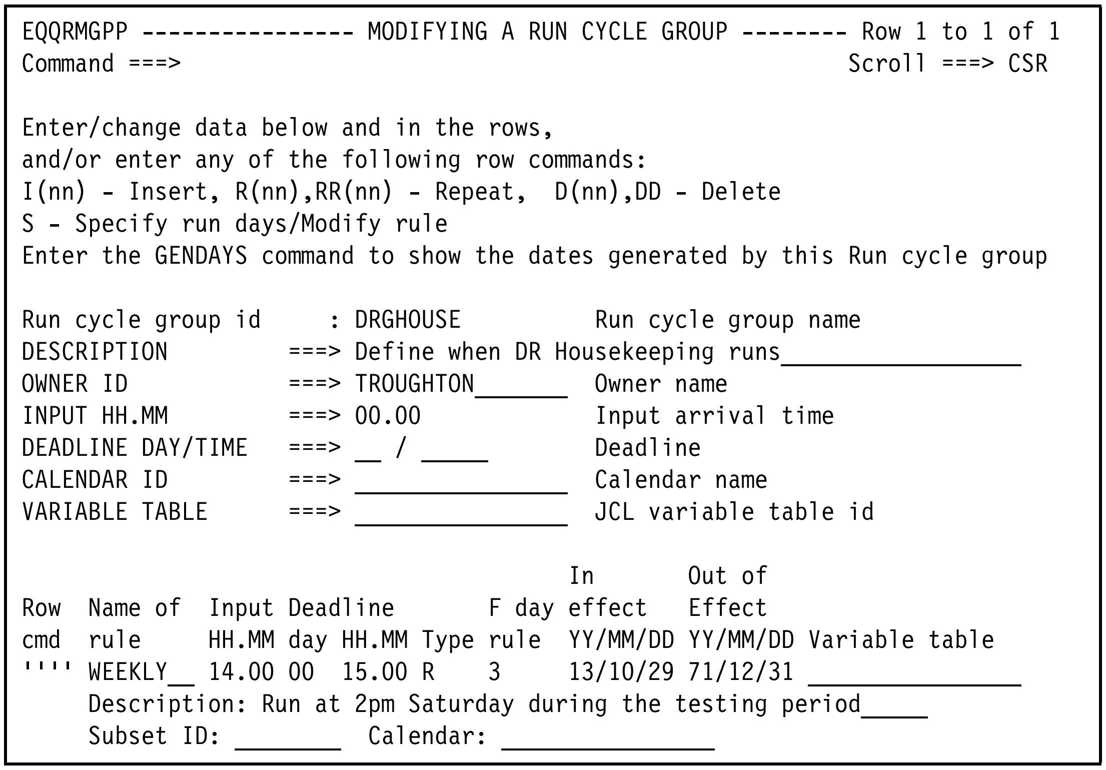

```
EQQRMGPP ---------------- MODIFYING A RUN CYCLE GROUP -------- Row 1 to 1 of 1
Command ===>                                                  Scroll ===> CSR

Enter/change data below and in the rows,
and/or enter any of the following row commands:
I(nn) - Insert, R(nn),RR(nn) - Repeat,  D(nn),DD - Delete
S - Specify run days/Modify rule
Enter the GENDAYS command to show the dates generated by this Run cycle group

Run cycle group id     : DRGHOUSE          Run cycle group name
DESCRIPTION         ===> Define when DR Housekeeping runs__________________
OWNER ID            ===> TROUGHTON_______  Owner name
INPUT HH.MM         ===> 00.00             Input arrival time
DEADLINE DAY/TIME   ===> __ / _____        Deadline
CALENDAR ID         ===> ________________  Calendar name
VARIABLE TABLE      ===> ________________  JCL variable table id

                                          In       Out of
Row  Name of  Input Deadline       F day effect   Effect
cmd  rule     HH.MM day HH.MM Type rule  YY/MM/DD YY/MM/DD Variable table
'''' WEEKLY__ 14.00 00  15.00 R    3     13/10/29 71/12/31 ________________
     Description: Run at 2pm Saturday during the testing period_____
     Subset ID: ________  Calendar: ________________
```

*Figure 5-36 Example of a runcycle group*

This example can then be referenced by any application definition, as shown in Figure 5-37.

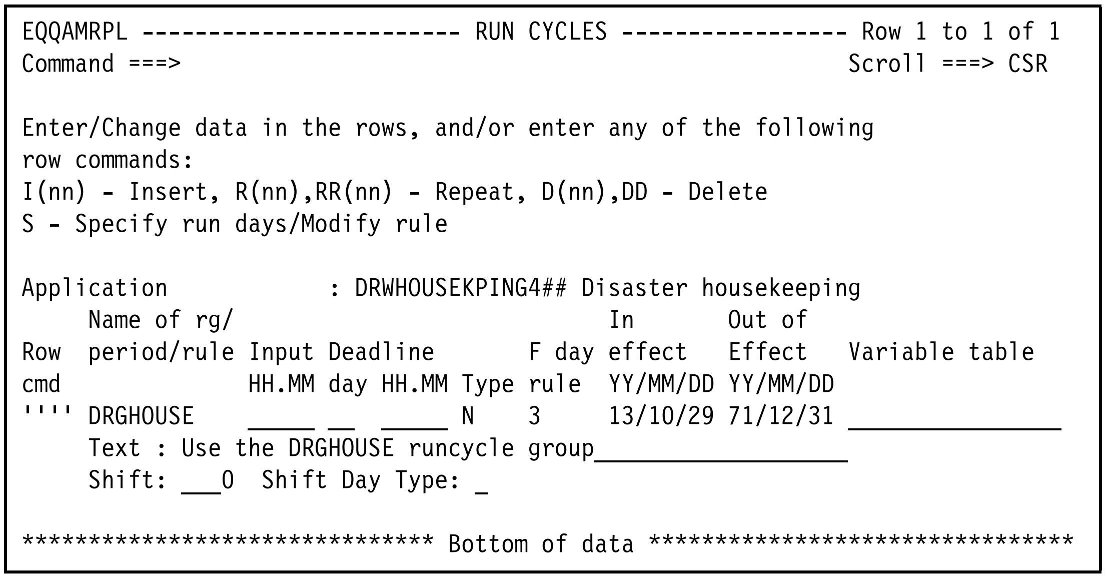

```
EQQAMRPL ------------------------ RUN CYCLES ----------------- Row 1 to 1 of 1
Command ===>                                                  Scroll ===> CSR

Enter/Change data in the rows, and/or enter any of the following
row commands:
I(nn) - Insert, R(nn),RR(nn) - Repeat, D(nn),DD - Delete
S - Specify run days/Modify rule

Application           : DRWHOUSEKPING4## Disaster housekeeping
     Name of rg/                             In       Out of
Row  period/rule Input Deadline       F day effect   Effect   Variable table
cmd              HH.MM day HH.MM Type rule  YY/MM/DD YY/MM/DD
'''' DRGHOUSE    _____ __  _____ N    3     13/10/29 71/12/31 ________________
     Text : Use the DRGHOUSE runcycle group___________________
     Shift: ___0  Shift Day Type: _

******************************* Bottom of data ********************************
```

*Figure 5-37 Referencing a runcycle group*

Any changes to the runcycle group automatically are reflected in all application definitions that refer to it, regardless of how many instances are involved.

## Scheduling object templates

The section “Using the Job-Library-Read exit (EQQUX002)” on page 44 describes a mechanism by which only one version of a job must be created and maintained regardless of the number of instances being used.

The scheduling techniques that have been described so far all require that a specific scheduling object is created for each individual instance. This is true for most workload automation products for static planned workload, but where dynamic insertion occurs, there are options to create a single scheduling object template and use it many times.

In the case of Tivoli Workload Scheduler for z/OS, the SOE toolkit dynamically can create an occurrence in the current plan from the batch loader by using a text syntax that can be used to define an application definition. This ability, with the ability to create and use variables and to include command text from other members, allows the creation and usage of templates in a flexible manner.

### *Creating an application template*

The first step is to create a single instance of the application definition you want to turn into a template in the Tivoli Workload Scheduler for z/OS database as normal. Then, run an SOE job to extract the batch loader for that application definition.

Figure 5-38 shows an SOE job that extracts the batch loader text for application definition DRWHOUSEKPING4#A. The statement **`OPTIONS STRIP(Y) SHOWDFLT(N)`** removes all trailing spaces and omits any keywords that are set to default values to reduce the amount of output.

```
//RUNSOE   EXEC EQQYXJCL,
//         SUBSYS=T91C
//OUTBL    DD SYSOUT=*,LRECL=72,RECFM=FB
//SYSIN    DD *
OPTIONS STRIP(Y) SHOWDFLT(N)
INCLUDE EQQFILE(AD)
SELECT AD ADID(DRWHOUSEKPING4#A)
```

*Figure 5-38 Batch loader extract*

The batch loader is output to the **OUTBL DD** statement, which is set to 72 characters so that it flows neatly in the first 72 characters of JCL. The output looks something like the example that is shown in Figure 5-39. This output should be stored within a library member in preparation for being converted into a template.

```
ADSTART ADID(DRWHOUSEKPING4#A) ADVALFROM(131101)
   DESCR('Disaster weekly instance') OWNER(GRANT) ODESCR('Richard E')
ADOP WSID(NONR) OPNO(001) JOBN(ZFIRST) DESCR('Start of application')
   DURATION(1)
ADOP WSID(CPU1) OPNO(005) JOBN(DRWHOINA) DESCR('Insert instance')
   DURATION(1) USEXTNAME(Y)
ADDEP PREWSID(NONR) PREOPNO(001)
ADOPEXTN EXTNAME(ALT=DRWHOIN@)
ADUSF UFNAME(USRJX) UFVALUE(SKL)
ADOP WSID(NONR) OPNO(255) JOBN(ZLAST) DESCR('End of application')
   DURATION(1)
ADDEP PREWSID(CPU1) PREOPNO(005)
```

*Figure 5-39 Batch loader output*

The batch loader includes the setting of extended job name (**ADOPEXTN**) and user fields (**ADUSF**) to use the Job-Library-Read exit.

The output itself needs a little amendment to turn it into a template. The following changes should be applied:

- The **ADVALFROM** keyword can be removed (this applies only to objects in the database).
- **ACTION(SUBMIT)** should be added to the **ADSTART** statement to cause the occurrence to be created directly in the current plan.
- **IA(!YMD1!OHHMM)** should be added to **ADSTART** to ensure that the dynamic occurrence has an input arrival time that mirrors the submitting occurrence.
- The last character of the **ADID** keyword should be replaced with **!TINST** to give the occurrence a unique name that matches the instance character.
- The last character of any **JOBN** keywords that apply to split jobs should also be replaced with **!TINST**.

Figure 5-40 shows how the batch loader looks after the conversion to a template.

```
ADSTART ADID(DRWHOUSEKPING4#!TINST) ACTION(SUBMIT) IA(!OYMD1!OHHMM)
   DESCR('Disaster weekly instance') OWNER(GRANT) ODESCR('Richard E')
ADOP WSID(NONR) OPNO(001) JOBN(ZFIRST) DESCR('Start of application')
   DURATION(1)
ADOP WSID(CPU1) OPNO(005) JOBN(DRWHOIN!TINST) DESCR('INSERT INSTANCE')
   DURATION(1) USEXTNAME(Y)
ADDEP PREWSID(NONR) PREOPNO(001)
ADOPEXTN EXTNAME(ALT=DRWHOIN@)
ADUSF UFNAME(USRJX) UFVALUE(SKL)
ADOP WSID(NONR) OPNO(255) JOBN(ZLAST) DESCR('END OF APPLICATION')
   DURATION(1)
ADDEP PREWSID(CPU1) PREOPNO(005)
```

*Figure 5-40 Batch loader that is converted to an application template*

### Using an application template

An application template can be used in a scenario that is similar to Figure 5-32 on page 58, in that the template is used for direct insertion instead of inserting an application definition from the database.

The job to do this task is similar to the direct insertion scenario. Figure 5-41 shows how to insert a template.

```
//RUNSOE   EXEC EQQYXJCL,
//         SUBSYS=T91C
//ADSKELS  DD DISP=SHR,DSN=DR.ADSKELS.PARMLIB
//SYSIN    DD *
VARSUB SCAN

VARSET TINST VALUE(A)
INCLUDE ADSKELS(DRSKEL)
MODIFY CPOP OPNO(001)
INSERT CPPRE PREADID(!OADID) PREIA(!OYMD1!OHHMM) PREOPNO(5)
MODIFY CPOP OPNO(255)
INSERT CPSUC SUCADID(!OADID) SUCIA(!OYMD1!OHHMM) SUCOPNO(200)

VARSET TINST VALUE(B)
INCLUDE ADSKELS(DRSKEL)
MODIFY CPOP OPNO(001)
INSERT CPPRE PREADID(!OADID) PREIA(!OYMD1!OHHMM) PREOPNO(5)
MODIFY CPOP OPNO(255)
INSERT CPSUC SUCADID(!OADID) SUCIA(!OYMD1!OHHMM) SUCOPNO(200)
```

*Figure 5-41 Inserting a template into the current plan*

In this example, the template is created as member DRSKEL in library DR.ADSKELS.PARMLIB.

The two core differences between this and the previous direct insertion technique is that a new variable is created that is called TINST, which contains the instance character for each insertion, and the **`INSERT CPOC`** is replaced with the **`INCLUDE ADSKELS(DRSKEL)`**.

What the **`INCLUDE`** does is process the batch loader statements to build an application definition record in storage, and then the **`ACTION(SUBMIT)`** keyword causes the newly constructed record to be introduced to the current plan through an **`INSERT CPOC`** command without passing an **`ADID`** argument. This causes the PIF to use the record in storage, rather than use one from the database.

Without SOE, building an entire application record from scratch in storage can be difficult, so an alternative approach is to build a single model instance in the database and then write a PIF program to perform the following actions:

1. Issue a **`SELECT AD`** command to retrieve the model instance in the database.
2. Step through the record in storage to overlay the last character in the application name, and any split job names with the instance character.
3. Issue an **`INSERT CPOC`** with no arguments to use the record that is constructed in storage.

The rest of the program works the same as the previous direct insertion scenario to resolve dependencies.

With this combination of the Job-Library-Read exit and SOE, it is possible to create a single JCL skeleton and a single application template, and then reuse them for any number of instances you might need, without having to create any additional operational elements.

If the scenario is a dynamic split, then you can also use this technique by making the process that decides how many instances to run create the SOE control statements, repeating the **`VARSET TINST`** to **`INSERT CPSUC`** block of statements once for each instance, but incrementing the value for TINST for each block.

### 5.3.6 Reducing operational impact

Taking a single job and splitting it into multiple jobs running concurrently increases demands upon the operational environment, such as increased consumption of initiators, storage, and processors. After the concept of splitting jobs becomes the normal approach on a system, many resource bottlenecks might be exposed. Chapter 6, “Resource management considerations” on page 71 describes the kind of resources that must be considered and provided for, but these resources are not finite and should be considered when designing the workflow.

Conditional execution and dynamic splitting go some way to limiting the usage of resources, but other mechanisms might need to be considered. By simply submitting all the work to JES when dependencies are met, they are served by JES in the order they are given to it. The workload that is controlled by Tivoli Workload Scheduler for z/OS, when defined with quality data concerning estimated run times and deadlines, can calculate the latest start times for each operation, which is based upon the amount of time that is estimated to run the jobs to each critical deadline. With this information, the workload can prioritize submission to JES; whenever it is faced with a choice of multiple jobs ready for submission, it favors the jobs with the earliest latest start time (LST).

With Tivoli Workload Scheduler for z/OS, there are two simple techniques that can be used. Other workload automation products can have similar features.

- Setting a parallel server limit per workstation. This technique stops the controller from allowing more jobs than the parallel server limit to run on each workstation at any one time. So, for example, having a workstation for each scheduled job class, and a parallel server count to match the number of available initiators, the controller submits only jobs when there are initiators available. In Tivoli Workload Scheduler for z/OS V9.2, up to 65535 parallel servers can be defined for a standard workstation that targets a single LPAR, and the same for a virtual workstation that can target multiple LPARs.
- Creating special resources. A special resource is a virtual concept. You define a name for each resource, tell the controller how many of them can be in use at once, and then add references to these resources to each job you want to restrict, letting it consume a portion of the resource exclusively. Then, like parallel servers, the controller allows only as many as you have defined to be in use at once, and will not allow a job to be submitted if there is no spare capacity for each resource. You can map resources to initiator classes and consume a count of 1, or any other type of resource, for example, CPU, and consume a count proportional to how much of the resource the job is consuming.

Both parallel servers and resources can have their quantities defined with different levels for different times of the day, for each day of the week or specific dates. Figure 5-42 on page 67 shows this concept in action. In this example, there is a class of jobs that should never have more than five jobs running simultaneously to protect from over-subscription of system resources. They are prevented from doing so by requiring a special resource with a total availability of five.

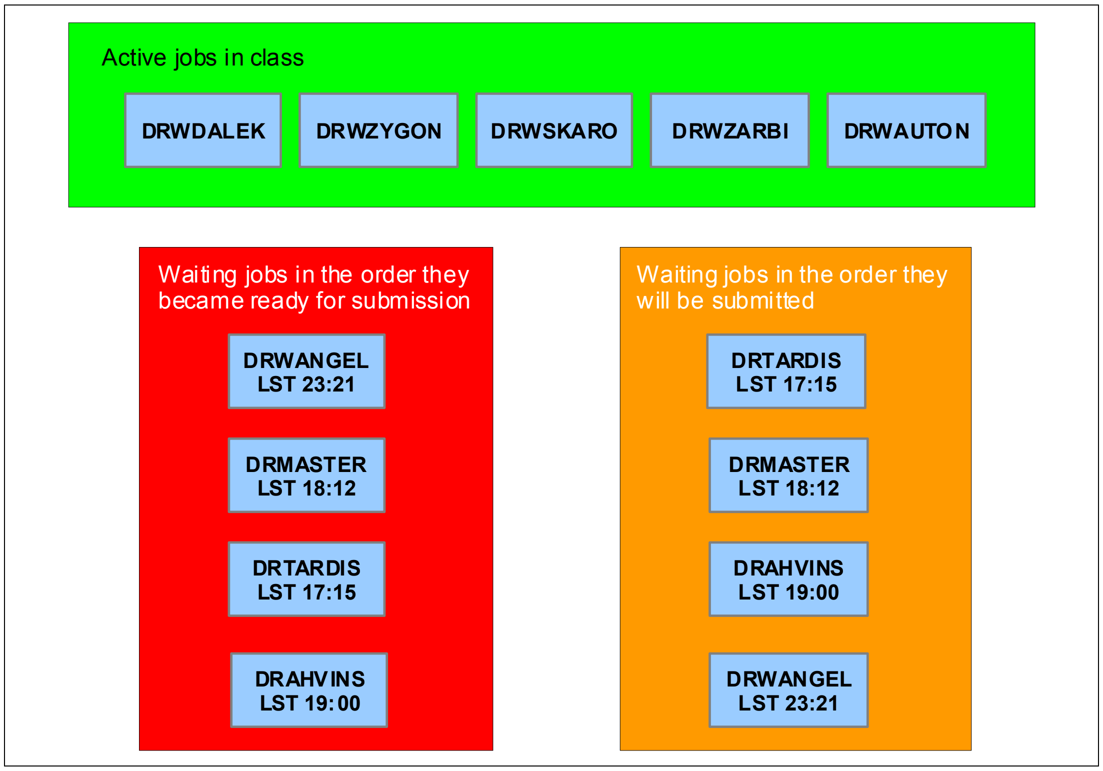


*Figure 5-42 Comparing job wait queues*

While five jobs are still running, the predecessors for DRWANGEL are completed, so DRWANGEL becomes eligible for submission, and has an LST of 23:21, which means the job must start by 23:21 to have a chance of deadlines being met. The job is held by the controller because the five running jobs are holding all of the special resource.

While the five jobs are still running, three more jobs become eligible for submission: DRMASTER (LST 18:12), DRTARDIS (LST 17:15), and DRAHVINS (LST 19:00).

When one of the five jobs completes, the controller looks for the earliest LST in the queue, so DRTARDIS is submitted next. Despite being the third to be ready, it wins because it has the earliest LST. DRWANGEL is forced to wait until no other jobs are around before it can move into a running state.

Had these jobs been submitted to JES when they were ready, they would have run in the order they were submitted. In this case, the most urgent job would have to wait for two less urgent jobs to complete before running, possibly missing its deadline.

## 5.4 Recovery and rerun

Recovery and rerun are dependent entirely upon the design of the application. In many ways, the concepts for parallel batch are no different from normal batch, but the concepts must be considered when splitting jobs for parallel processing.

For a normal job, before being split, there are typically two mechanisms that are considered for recovery and rerun:

- Restore to start of job, which involves restoring all data to the same state as it was before the job started, or rolling back data to the initial state.
- Checkpointing, which involves tracking the latest committed record in the input data that is processed successfully. If there is a failure, any uncommitted updates are rolled back to the latest checkpoint and input processing resumes from the checkpoint.

With a single unsplit job, there usually are two restart points to consider at the design stage: from the start of the failing step or the start of the failing job. With a job split into parallel instances, there are more start points to consider:

- Each individual instance must be able to be recovered and restarted separately, without impacting the others.
- In some cases, depending on the style of the split, it might be necessary to restart from the beginning of all parallel instances.
- Additionally, depending on the nature of the error, the start node might require rerunning to redrive the split process after correction of the error.

Many workload automation products have automated recovery features that automatically can start recovery actions in circumstances where the problem is understood. Figure 5-43 shows an example of automated recovery in Tivoli Workload Scheduler for z/OS.

```
//*%OPC SCAN
//&OJOBNAME JOB (99999),'DR W HKP',USER=HURT,
//         CLASS=A,MSGCLASS=X
//*%OPC SETVAR TINST=SUBSTR(&OJOBNAME,8,1)
//*%OPC RECOVER JOBCODE=(666),ADDAPPL=KEY2TIME,RESTART=Y
//*
//         EXEC DRWHOUSE,TINST=&TINST,MIN=&MIN,MAX=&MAX
//
```

*Figure 5-43 Automatic recovery*

The **RECOVER** statement identifies return code 666 as a known error condition for this job, for which the recovery is to add a predefined application description as a predecessor to the failed job before rerunning the failed job. In this instance, the return code is known to be a contention and all that KEY2TIME does is wait for 30 seconds, effectively allowing the job to be tried again automatically 30 seconds after it fails. For more information about the **RECOVER** directive, see *Workload Scheduler for z/OS Managing the Workload*, SC32-1263.

All of these points must be considered. Where appropriate, recovery processes should be designed and documented accordingly. Figure 5-44 on page 69 shows the possible restart points that require assessing. They are indicated by the red bars to show the scope of each restart point.

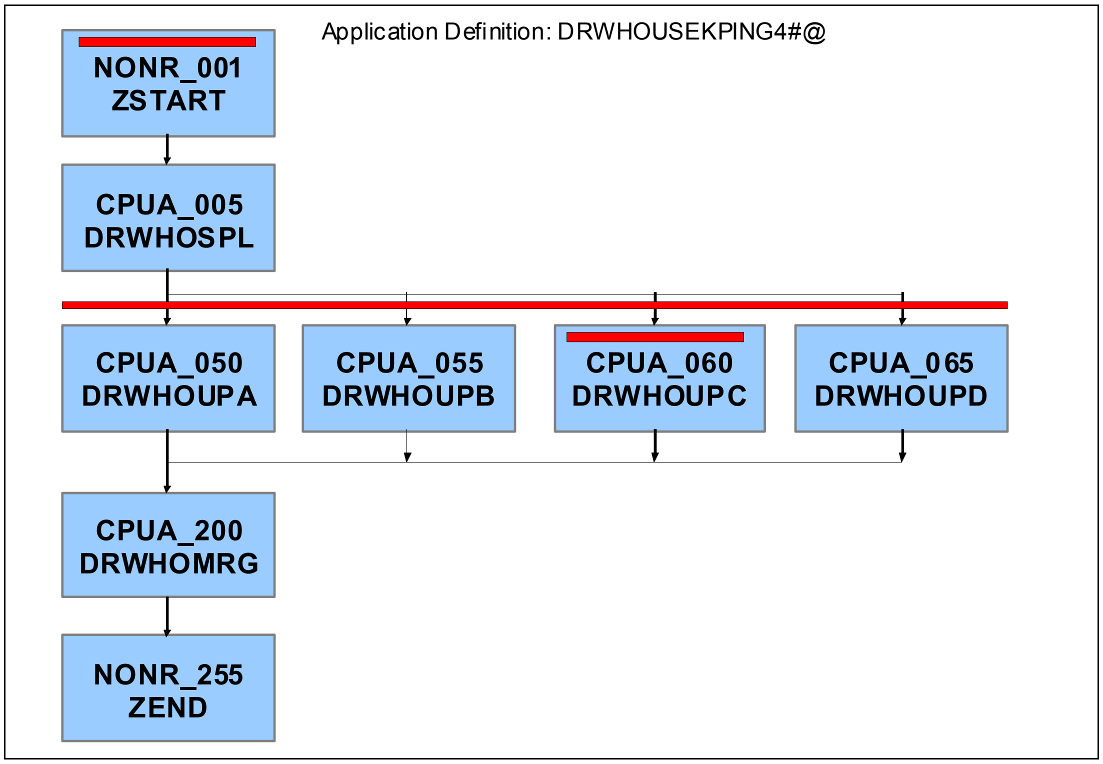


*Figure 5-44 Possible restart points*

## 5.5 Workload service assurance

After a job is split into many parallel instances, there are more moving parts to track. Thus, proactive monitoring becomes more important to watch for problems and potential delays.

### 5.5.1 Milestone alerting

A common process for service assurance is to pick important jobs in the batch and schedule check jobs at key points to see whether that job reached the expected status by that point. This typically uses batch utilities for the workload automation product concerned so that they give a nonzero return code, fail, or raise an alert in a monitoring tool.

The main issue with milestone alerting is that it issues only alerts when the individual jobs being monitored have an issue. It does not warn of potential problems in other jobs until you reach the milestone checkpoint. Another issue is that network flows, durations, and criticality change over time, but the milestones are rarely altered to match these changes.

### 5.5.2 Critical path monitoring

Some workload automation tools have a more dynamic approach to workload service assurance as a built-in feature. The concept behind monitoring the critical path is to provide the workload automation tool with enough information for it to spot problems without being concerned only with individual milestones.

Tivoli Workload Scheduler for z/OS has such a feature, where it knows the estimated durations of each job and the relationships between them, so after a critical endpoint is marked in the workload, and given a business deadline, it can calculate the LST of every operation leading up to that endpoint.

With that knowledge, Tivoli Workload Scheduler for z/OS can spot when any job on the route to the endpoint does not start in time to reach the deadline. It warns of errors that occur on a route to critical end points and focuses operators on which failed jobs need urgent attention.

Tivoli Workload Scheduler for z/OS also escalates the priority of jobs within Tivoli Workload Scheduler for z/OS and raises the service class of jobs on critical paths that are running behind schedule to get things back on schedule.

This approach can lead to notification of potential problems earlier than milestone alerting, but relies upon good data to make those predictions. So, it is important to examine what critical path monitoring tools to which you have access, and focus on getting procedures in place for accurate provision and maintenance of your estimated job durations to keep a closer eye on the increased job volumes that occur as a result of parallel batch.

# 6 Resource management considerations

This chapter describes resource management considerations, which usually are the responsibility of performance and capacity planning specialists in the system programming department.

Increasing parallelism by splitting jobs changes the pattern of resource usage, whether logical (such as initiators) or physical (such as processors). This chapter describes some of these considerations, and they are also described in *Batch Modernization on z/OS*, SG24-7779, but this chapter relates specifically to job splitting.

## 6.1 Processors

Whether individual jobs use more processor cycles or not, increasing parallelism tends to create bursts (or spikes) of higher processor usage. In some cases, these spikes cause processor queuing, which can delay both the jobs being split and other work. Processor queuing can be managed in three main ways:

- Reduce the jobs' usage of the processors through tuning.
- Provision more processors, perhaps with processor upgrades.
- Manage processor provisioning within the logical partition (LPAR) by using Workload Manager (WLM).

These topics are beyond the scope of this publication but are used by many installations.

## 6.2 Memory

When you run more jobs in parallel, the usage of memory is likely to increase. If usage increases too much, the system might page. Paging causes delays in work, so it is important to manage memory carefully through the batch window.

If you split a job, the memory requirement might increase significantly or only a little. Here are some of the factors that govern whether the requirement increases:

- If the program acquires significant working storage, it is replicated. An example of this is an array.
- The program itself usually is loaded into memory.
- If the program has significant buffer pools of its own, they are replicated. Examples of this are Virtual Storage Access Method (VSAM) local shared resources (LSR) buffer pools and Queued Sequential Access Method (QSAM) buffers.
- If the program starts **DFSORT** to perform operations involving sorting, the sort workspace in memory is replicated.
- If the program uses DB2 data, the DB2 buffer pools are shared between all users and not replicated. Under most cases, the buffer pool access patterns are improved, so additional DB2 buffers are not required.

### 6.2.1 Concurrent sorting operations

Sorts usually are low in processor usage but high in workspace requirements and are I/O-intensive. **DFSORT** uses a combination of memory and disk space for workspace. **DFSORT** calls the z/OS STGTEST SYSEVENT service several times in the course of a sort. This service returns information about how much memory a **DFSORT** sort can use for workspace.

Two or more programs calling STGTEST SYSEVENT at roughly the same time probably receive the same answer because the service is not designed to actively manage memory sharing between callers. For example, eight concurrent sorts might call STGTEST SYSEVENT and each receive the answer "you can use 1 GB of memory". This does not mean 8 GB of memory is available at that instant but that only 1 GB is. In this example, if each sort used 1 GB, this causes a massive overcommitment of memory.

A `DFSORT` sort minimizes the risk of memory overcommitment by calling STGTEST SYSEVENT several times during the sort, as do its peers. By doing this, a sort can adjust its usage of memory more dynamically than if it just called STGTEST SYSEVENT once at the beginning of the sort. Because it is more dynamic, the risk of overcommitment is much reduced. The algorithm in z/OS Version 2 Release 1 `DFSORT` is enhanced to further reduce the risk of overcommitment.

Despite concurrent sorts being unlikely to cause overcommitment, they can cause a large spike in memory usage. Such a spike might cause a shortage of memory for other jobs.

## 6.3 Disk

Both disk space and disk I/O bandwidth can be affected by splitting jobs. There are two major concerns:

- Instantaneous I/O access density and disk space requirements can be much higher when the job is split. For example, a VSAM lookup file might be accessed more intensively when shared by several job instances.
- More transient data sets are likely to be created. For example, splitting an input transaction file to feed the instances creates a second copy of the data.

## 6.4 Tape

If the job uses tape, it needs tape drives, whether real or virtual, to mount the tape volumes. Unless splitting the job reduces the requirement to use tape, the concurrent tape drive requirement greatly increases.

When splitting a tape job, consider re-engineering it to avoid the usage of tape. For example, the tape data set might be transient and suitable for replacing with a BatchPipes/MVS pipe.

## 6.5 Initiators

If the split application has too few initiators to run concurrently, some jobs are delayed while waiting for an initiator, which can negate some of the benefit of running split jobs. It is important to manage the degree of requested parallelism and the number of initiators carefully to avoid this conflict.

SMF type 30 Subtype 5 records can be used to diagnose cases where jobs are not immediately initiated, whether because of a shortage of initiators or because of other reasons.

If you need initiators on a short-term basis, consider whether the Job Entry Subsystem 2 (JES2) startjob ($SJ) command can be used to create temporary initiators. Issuing this command immediately initiates a job and terminates the initiator immediately after the job has run.

### 6.5.1 WLM managed initiators

With WLM managed initiators, WLM is relatively slow to create new initiators. If, for example, 32 jobs are started at once, it takes at least a few seconds for WLM to create additional initiators to run all these jobs. Do not rely on WLM to create large numbers of initiators in a short period.

WLM assesses system conditions when deciding whether to create initiators. Although your split application might need more initiators to run, WLM might decide not to create them for you. From the overall batch workload and system point of view, this might be the correct decision, even if it is not good for the application. A good example of this is where the processor is too busy to support additional work.

7

# Case study

This chapter describes a small case study that illustrates some of the techniques for making splitting more scalable, and how to use instrumentation to guide you in refining your implementation. This chapter describes our case study scenario, and what modifications were made and how they affected performance and scalability.

This case study was not performed to a benchmark standard, but is an informal illustration of the techniques and their effects.

## 7.1 Case study scenario

The authors of the book wrote a simple application. This application is a single batch COBOL program that accesses DB2, performing complex read-only queries, then updating many rows and producing a simple report.

For each case, we ran the program the following ways:

- “0-up”: Unmodified for splitting.
- “1-up”: Prepared for splitting but not actually split.
- “2-up”: Split into two instances.
- “4-up”: Split into four instances.
- “8-up”: Split into eight instances.
- “16-up”: Split into 16 instances.
- “32-up”: Split into 32 instances.

The purpose was to measure the effectiveness and cost of splitting the work as the number of instances grew. Effectiveness was measured by using two main metrics:

- Elapsed time of the longest-running instance.
- The total CPU that is consumed across all instances.

### 7.1.1 Application synopsis

The application that we wrote was intended to be an example of a common processing scenario that is likely to be present in many systems. The COBOL program reads two input files and seven DB2 tables, updates one DB2 table, and generates a report. The input file lists the key values to be processed. These key values correspond to descriptive values in a Virtual Storage Access Method (VSAM) lookup file.

The original program is described in this section followed by the modifications that were made to both the JCL and COBOL code so that multiple instances can run in parallel.

#### Summary of original program logic

The original program is called POKB302 and can be summarized as follows. For each record in the input file, the following actions occur:

- DB2 SQL statements run.
- The descriptive value is obtained from the VSAM file data.
- A report record is written.
- DB2 UPDATEs occur.

Figure 7-1 on page 77 is included for illustrative purposes and is not meant to be prescriptive of the code that is suitable for splitting. It shows in more detail the high-level logic of the original program.

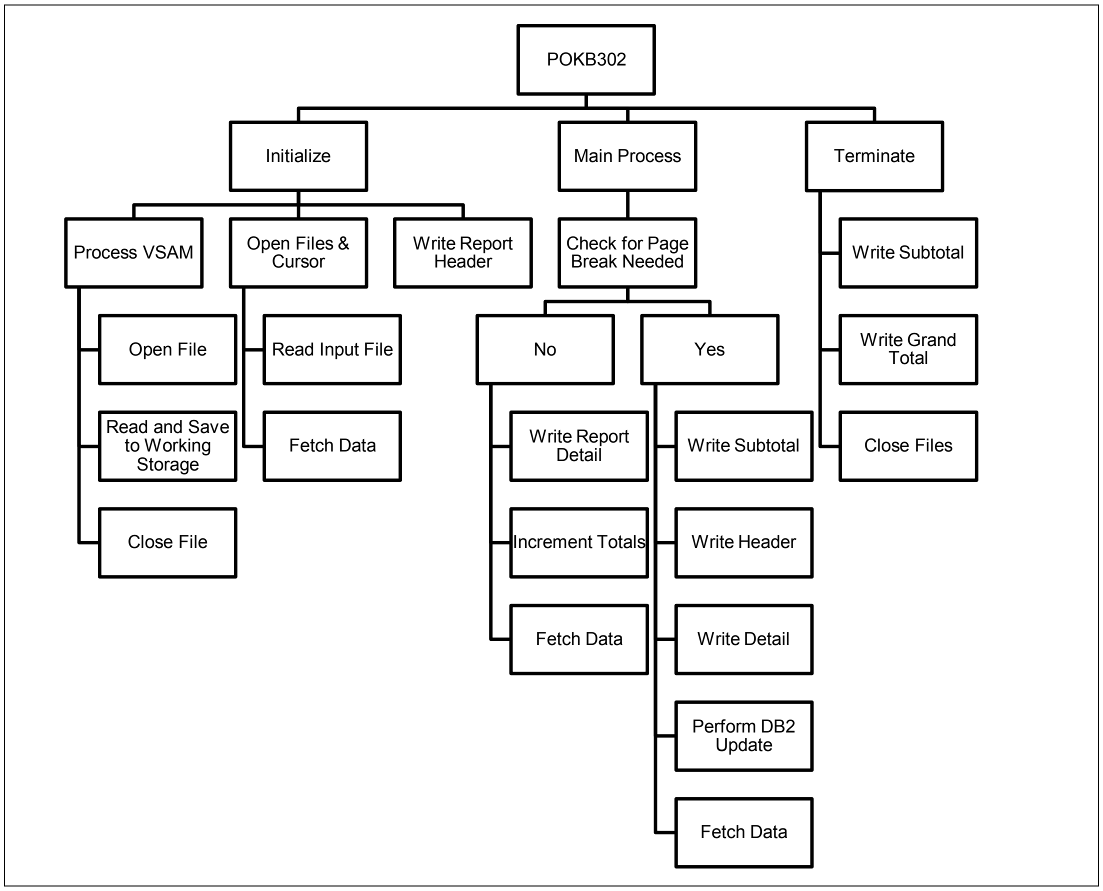


*Figure 7-1 High-level logic of program POKB302*

In the runs that are described in this chapter, "0-up" refers to this original program, running in whatever environment on which the test was conducted. For example, when the table is partitioned, 0-up refers to the original program running against the newly partitioned table.

### *The report*

Most of the content of the report is irrelevant to the splitting process. Some elements are included in our example to highlight examples of where the report content requires some consideration. In our example, the report consists of the following elements:

- Headers
- Detail lines

  The detail lines include the descriptive data from the VSAM file and calculated values, including summation, and average, minimum, and maximum values.
- Subtotals and a page throw at every change of the first two fields on the report

  These mathematical elements of the report are included deliberately to highlight that part of the process of splitting the work that might require additional data to be maintained that was not necessary in the original application.
- Grand total line

### *The VSAM file*

In our example, the VSAM file is a simple data set containing a key and a corresponding description value. The driving input file contains a number of records of key data and the description is used in the report. The VSAM data is read in the initialization phase of the program and the key and description pairs are stored in working storage.

## Original JCL

The diagram in Figure 7-2 shows the JCL flow for the original program. It shows that the report is produced directly from the original program and that no other processing steps are required.

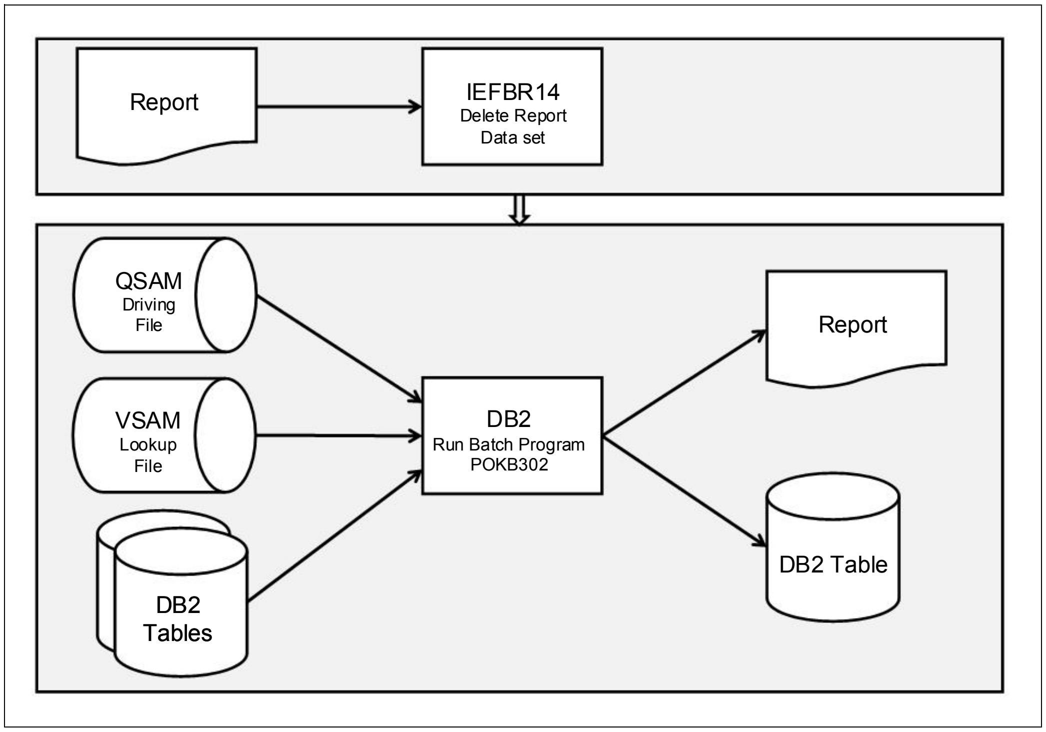


*Figure 7-2 JCL flow for the original program POKB302*

## JCL modifications

As shown in Figure 7-2, the original program was run in a single JCL job. This single job is split into four jobs to enable more than one instance of the program step to run at the same time. Here is a summary of the processing in the new jobs:

- Split the input file into a fixed number of files, each containing a subset of the data.

  In our example, we use a static split. Individual JCL jobs are created for 0-up, 1-up, 2-up, 4-up, 8-up, 16-up, and 32-up test cases. Each of the JCL jobs references a fixed number of files that are relevant to the test.

- Run the COBOL DB2 program POKB303.

  POKB303 is a copy of the original program with modifications for the split process, as described in “Update job” on page 80.

- Merge and sort the output files from program POKB303.

  In our example, the card dealer method of splitting the input file is used. Each instance of the update program processes data that was not sequential in the original run. This makes the sort necessary to ensure that the report is created in the same sequence as that produced by the original program.

If the sort is not necessary, then other processing options are available, such as concatenating the DD statements in the JCL.

- Run the COBOL program POKR004.

  A new program is needed to generate the report.

### *Fan-out or split job*

Figure 7-3 shows that a `DFSORT COPY` is coded to split the input file into as many files as there will be instances of the update job. Chapter 5, "Operational considerations" on page 29 describes options for controlling the split of the input file, whether statically or dynamically. In this case study, we use a static split so that there are a fixed number of files that are created by the job.

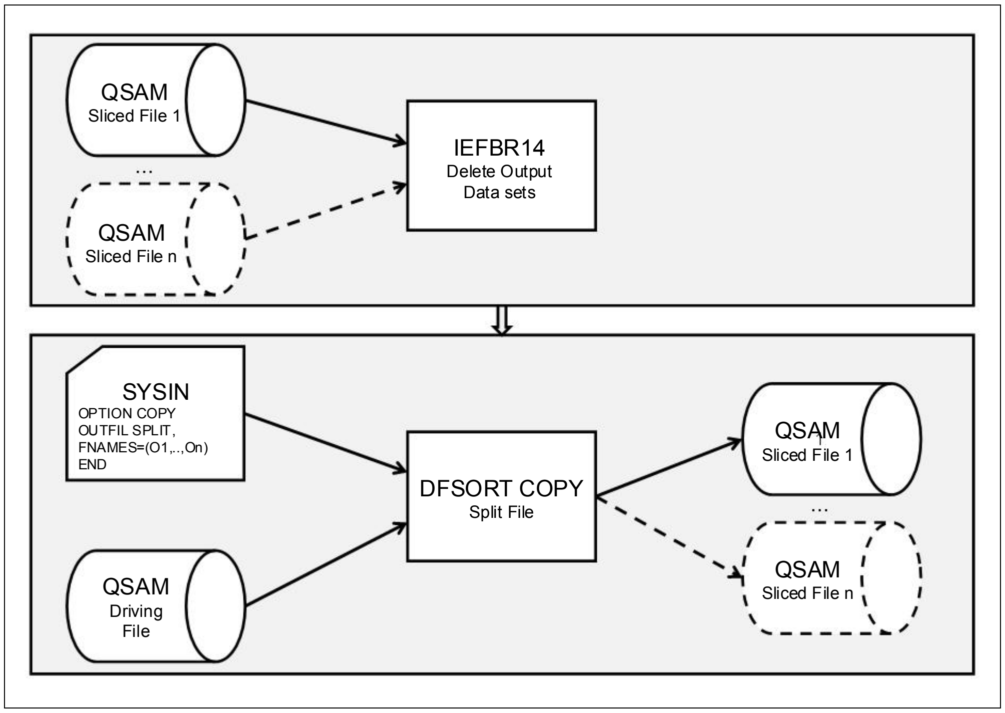


*Figure 7-3 The fan-out or split job*

The original report was sequenced in the ascending key order of the data in the QSAM Driving File. We use a card dealer split function to create multiple driving files, one for each instance of the update program, as described in "Card dealing" on page 18. This means that each instance of the update program processes data that was not sequential in the original run.

### Update job

The bulk of the processing that is done by the original program, POKB302, is now done in multiple update jobs. Each of these update jobs processes a subset of the original input file. The JCL flow for the update is shown in Figure 7-4.

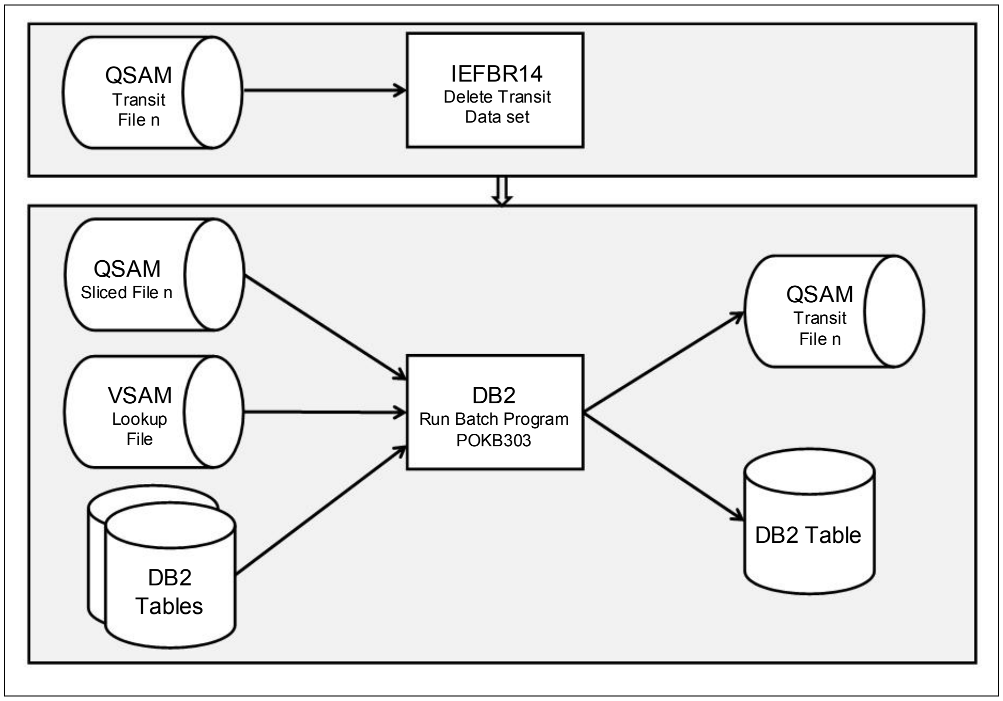


*Figure 7-4 The new update job running program POKB303*

The difference between the original JCL and the modified update job is that the new update program, POKB303, is being run and a new transition file is being created instead of the report.

In our example, the content of the transition file is the original report details, minus the header and totals. It is necessary to include additional data that was not on the report so that the total fields can be calculated properly in the new report program.

### Fan-in or merge job

When all of the instances of the update job run successfully, a fan-in or merge step is required to collate the multiple transition files. Figure 7-5 on page 81 shows the JCL flow for our example. In our case, a sort is also required to bring the data back into the correct sequence for the report.

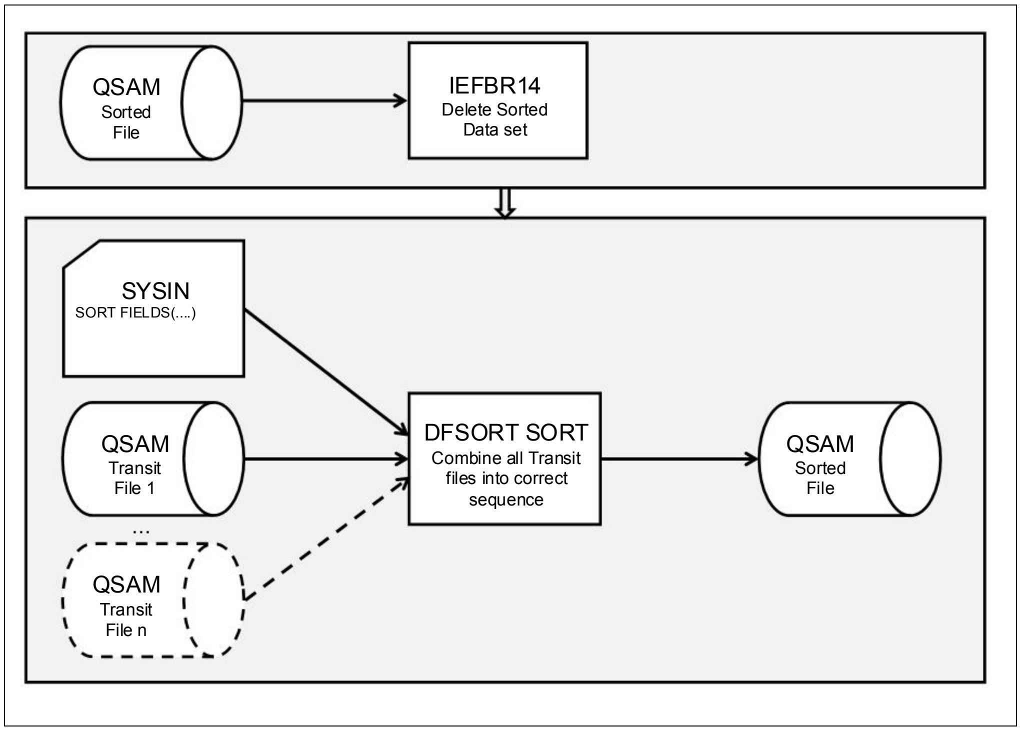


*Figure 7-5 The fan-in or merge job*

As mentioned in “Fan-out or split job” on page 79, our example uses a static split, so we also use a static merge job for each of the multiway instance tests, which means that there are a fixed number of files input to the SORT.

### *The report step*

Figure 7-6 shows the job flow for the final stage of creating the report. After the multiple transit files are merged and sorted into the correct sequence, a new program, POKR004, is created to add the reporting headers, subtotals, and grand totals.

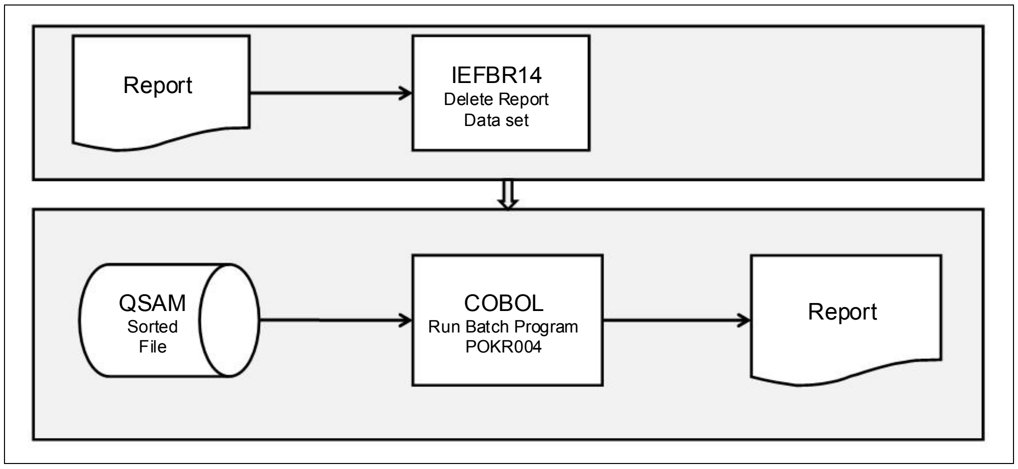


*Figure 7-6 The new report job running program POKR004*

We include an average in the subtotal and grand totals to illustrate that it might be necessary to pass additional data between the update program and the report program to maintain the accuracy and completeness of the reported data. In our example, the number of elements contributing to the average must be added to the transition file.

The new COBOL program, POKR004, reads the sorted data as input and creates the final report.

**Note:** You can use ISPF option 3.13 SuperCE to verify that the report from a multiple instance run is identical to the report that is created by the original program.

## COBOL program modifications

As mentioned in “JCL modifications” on page 78, this approach requires changes to the COBOL code. In our example, a new update program is created, POKB303, and the reporting functions are moved to a second program, POKR004.

### *New update program*

The original program is copied to POKB303 and the following changes are made:

- The reporting header processing is removed.
- The subtotal and grand total processing is removed.
- The report detail record write processing is replaced with a write to a QSAM output file.

  The extra data that is needed to maintain the reported averages is added to the record layout.
- DB2 COMMIT processing is added.

  If the original program does not have any DB2 COMMIT processing, it is likely that you must make more changes to allow correct restart processing in the case of a failure, for example, adding checkpoint processing to enable repositioning on the input file or on a driving cursor. This is outside the scope of our example in this book.

The high-level logic of POKB303 is shown in Figure 7-7. This diagram is included for illustrative purposes only.

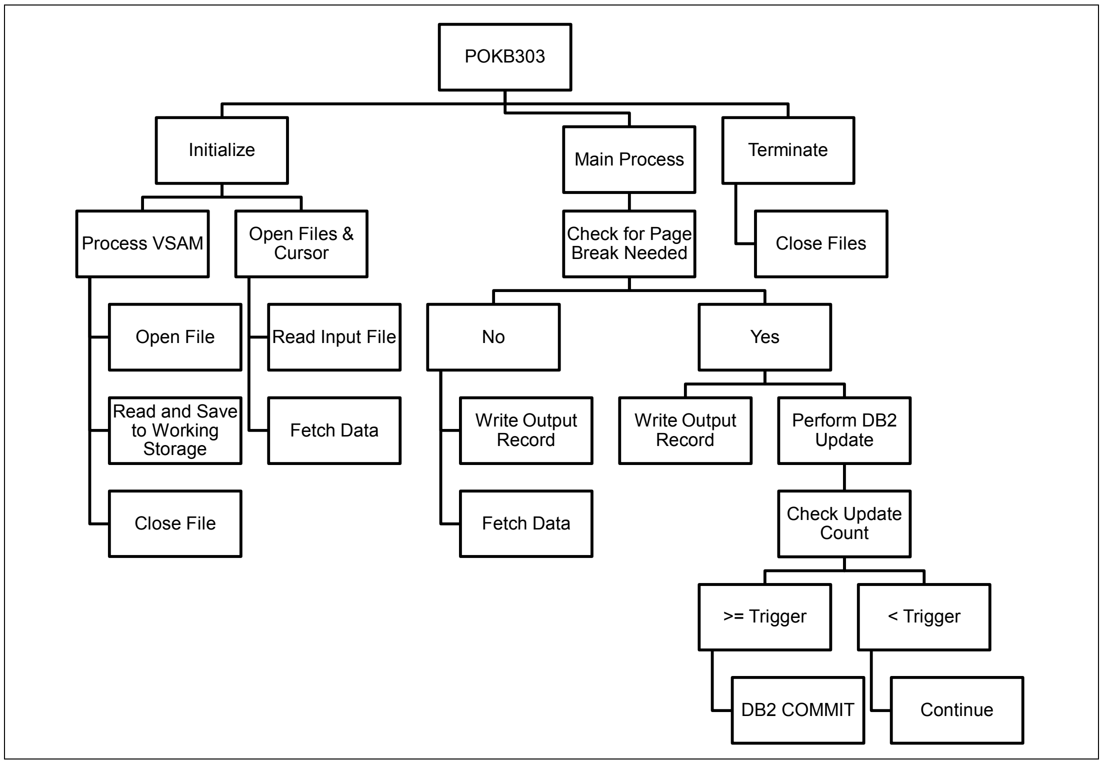


*Figure 7-7 High-level logic of program POKB303*

### New report program

The majority of the new report program (POKR004) logic is concerned with the addition of the report headers, sub totals, and grand totals. Logic is also included to use the additional data in the transit file to correct the calculation and reporting of the average values.

The high-level logic of the report program, POKR004, is shown in Figure 7-8.

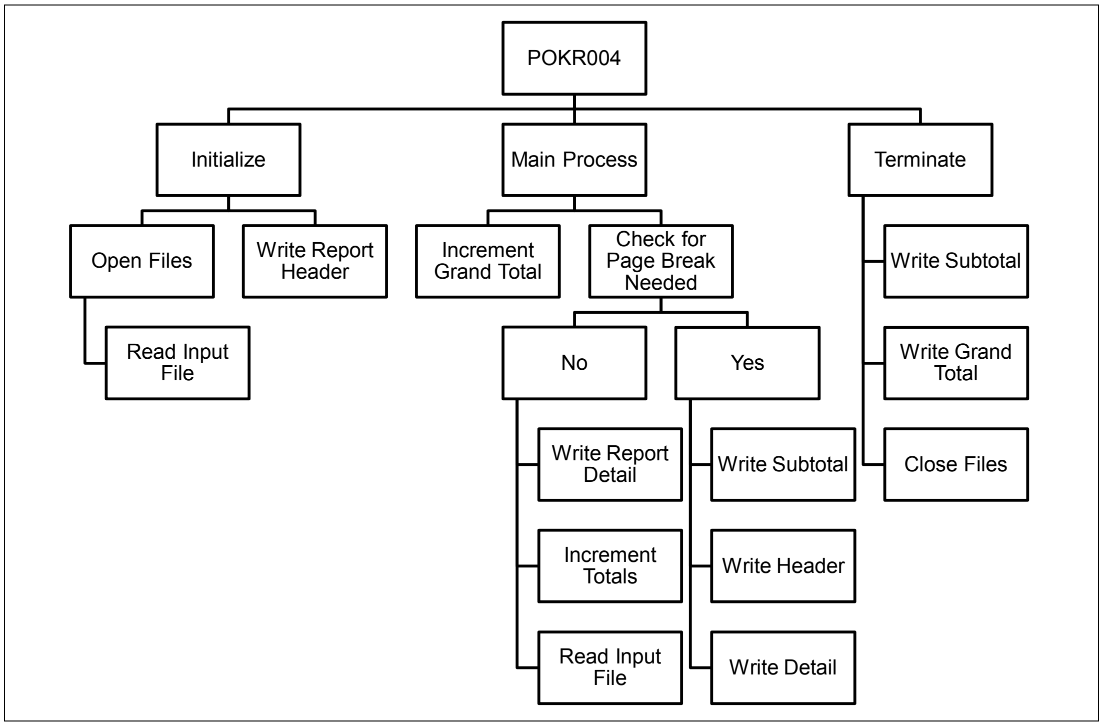


*Figure 7-8 High-level report program logic*

## Exclusions

In any scenario, there are many options for tuning an application. In our scenario, there are a many things that we do not do. Our focus is on showing improvements that can be gained by increasing the number of instances that are running in parallel. Many people might want to evaluate whether the improvements in elapsed time can be gained without having to resort to splitting batch jobs into multiple instances. We do not do the following tasks:

- Tune any of the SQL statements in the program code.

  We ensure that the access paths of the two programs are consistent by using the bind option `APREUSE(ERROR)`.

> **Note:** We might use the new bind option of `APREUSE(WARN)`. The `APREUSE` option specifies whether DB2 tries to reuse previous access paths for SQL statements in a package. DB2 uses information about the previous access paths from the directory to create a hint.
>
> With `APREUSE(ERROR)`, DB2 tries to reuse the previous access path for all statements in the package. If statements in the package cannot reuse the previous access path, the bind operation for the package ends.
>
> By specifying `APREUSE(WARN)`, DB2 ignores the previous access path for a statement that cannot reuse the previous access path and continues to generate a new access path.

An example of the messages from **APREUSE** is shown in Example 7-1.

*Example 7-1 APREUSE messages*

```
DSNT286I  -D1B3 DSNTBBP2 BIND FOR PACKAGE = DB1B.KW1.POKB303,
           USE OF APREUSE RESULTS IN:
           8 STATEMENTS WHERE APREUSE IS SUCCESSFUL
           0 STATEMENTS WHERE APREUSE IS EITHER NOT SUCCESSFUL
             OR PARTIALLY SUCCESSFUL
           0 STATEMENTS WHERE APREUSE COULD NOT BE PERFORMED
           0 STATEMENTS WHERE APREUSE WAS SUPPRESSED BY OTHER HINTS.
```

- Tune the DB2 environment over the changes that are stated specifically in this publication.

  The DB2 environment our tests run in is a new installation of DB2 11 with the default ZPARM values.
- Amend the program code to access specific partitions, where partitioning is applied to the tables. This is controlled by splitting the input file.

### 7.1.2 Environment

The tests are run on a zEnterprise EC12 processor. LPARs are defined so that the logical partition (LPAR) essentially has an entire book of processors that is dedicated to it. The environment is designed so that memory is unlikely to be constrained. For most tests, CPU is also unconstrained. The following major software products are used:

- z/OS Version 2 Release 1
- Job Entry Subsystem 2 (JES2) for z/OS Version 2 Release 1
- DB2 11 for z/OS
- IBM Tivoli OMEGAMON® XE for DB2 Performance Expert on z/OS Version 5 Release 1 Modification Level 1
- z/OS Version 2 Release 1 DFSORT
- Enterprise COBOL Version 5 Release 1
- Tivoli Workload Scheduler for z/OS Version 9 Release 1

### 7.1.3 Instrumentation

There are two main types of System Management Facility (SMF) instrumentation that is used in our example:

- The test job's elapsed time and CPU time is measured by using SMF 30 Subtype 5 (Job Termination) records. Individual step times and resource consumption are measured by using SMF 30 Subtype 4 (Step Termination) records.
- The job's DB2 activity is measured by using DB2 Accounting Trace (SMF type 101) with trace classes 1, 2, and 3 enabled. Although SMF 101 does not include step number, a job's DB2 processing is all in a single step, so the correct step can readily be identified. DB2 Accounting Trace is further described in "DB2 Accounting Trace" on page 85.

z/OS Version 2 Release 1 introduced variable blocked spanned (VBS) data set support in the Restructured Extended Executor (REXX) programming language. We use this support to write a REXX program with two functions:

- Write copies of the previously mentioned SMF type 30 and SMF type 101 records to a separate data set. The program selects the records, using the naming convention for the jobs in our test suite.
- Report major fields in the selected SMF type 30 records, for example, start and end time stamps, elapsed time, completion codes, and CPU time.

The REXX program is sufficient to extract the instrumentation that we need from the SMF type 30 records.

## DB2 Accounting Trace

For DB2 batch jobs, Accounting Trace is an important source of information. For detailed job tuning, turn on trace classes 1, 2, 3, 7, and 8. Trace classes 7 and 8 are needed only if your DB2 program uses packages. Our test case does not use packages.

Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS is used to format the SMF type 101 records. The Accounting Trace long report is used. Using the REXX program to limit the number of SMF type 101 records to those for the jobs in our test greatly reduces the run time and output of this report.

Use Accounting Trace to determine where the job spends its time:

- Subtract class 2 elapsed time from class 1 elapsed time to determine the time that is spent in the application program rather than DB2. This publication calls this time non DB2 time.
- Subtract class 2 CPU time from class 1 CPU time to establish the CPU time not in DB2. This publication calls this time non DB2 CPU time.
- Within class 2 elapsed time, a number of class 3 elapsed time components are recorded. For example, lock/latch wait time records time where the application waits for a DB2 latch or lock.
- Not-accounted-for time, within class 2 elapsed time, usually is time that is spent queuing for CPU.
- Within non DB2 Time, the portion that is not CPU usually is a combination of queuing for CPU and performing I/O.

We examine all of these time categories (or buckets) in our case study. The observations that we present are based on these buckets.

Accounting Trace is essential to our tuning effort because of the following reasons:

- It informs the team about which tuning action to take next.
- It explains most of the results that we obtain.

Use Accounting Trace in your tuning efforts.

Because we have no packages, we do not enable trace classes 7 and 8. If you have packages, then use trace classes 7 and 8 to accomplish the following tasks:

- Determine which packages use the bulk of the CPU.

  Use the list of large CPU packages to direct your CPU tuning efforts.
- Determine which packages contribute most to each of the significant Class 3 time components.

  Seek ways of reducing these large components.

In many cases, package analysis reduces the need to use more difficult techniques, such as analyzing Performance Trace or using Explain to diagnose SQL access paths. In other cases, it allows you to focus these techniques on a small subset of the job's SQL.

## 7.2 Test iterations

We start with the base case, which we deem a "naive" implementation of job splitting. Then, we modify our implementation and observe the behavior of the application at various levels of splitting.

Before every test iteration, the following tasks are done:

1. The updated DB2 table is reloaded to refresh the data and inline RUNSTATs are taken.
2. A BIND is done for each of the two COBOL DB2 programs.
3. The DB2 member is stopped and started.

No other activity is running at the same time as the test iterations.

**Note:** In all of the test runs, the CPU usage and Elapsed Time figures are taken from the JESMSGLG output of the job.

In all test scenarios, the fan-in and report jobs all report zero CPU and zero elapsed times. The fan-out job for 32-up reports zero CPU usage and 0.1 minutes elapsed time. For consistency with the base job, these are included in the figures that are used in the graphs.

Table 7-1 shows the test runs that are run along with the high-level findings for each one.

*Table 7-1 Test run summary*

| Test run | Description | High-level finding |
|---|---|---|
| 1 | ► All DB2 tables are non-partitioned<br>► No COMMIT processing | ► Cannot run more than 1 instance concurrently |
| 2 | ► All DB2 tables are non-partitioned<br>► COMMIT processing | ► Scales to 8 up successfully<br>► Increasing total CPU usage as number of instances increase |
| 3 | ► Updated DB2 table is partitioned 8 ways<br>► All other DB2 tables are non-partitioned<br>► COMMIT processing | ► Scales to 32-up successfully<br>► Overall reduction in total CPU and maximum elapsed time<br>► Flat CPU cost 1-up through 8-up |
| 4 | ► Updated DB2 table is partitioned 16 ways<br>► All other DB2 tables are non-partitioned<br>► COMMIT processing | ► Scales to 32-up successfully<br>► Overall reduction in total CPU and maximum elapsed time that is maintained<br>► Flat CPU cost 1-up through 8-up<br>► No discernible difference to 8 way partitioning |

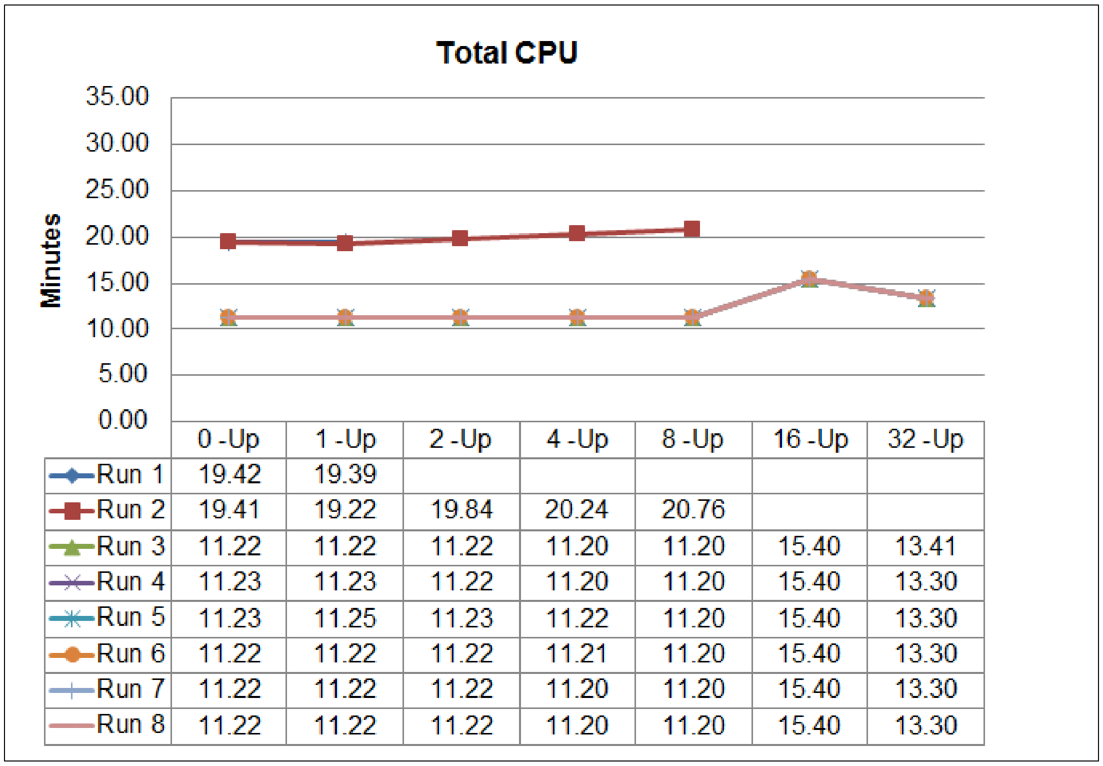
| Test run | Description | High-level finding |
|---|---|---|
| 5 | ► Updated DB2 table is partitioned 32 ways<br>► All other DB2 tables are non-partitioned<br>► COMMIT processing | ► Scales to 32-up successfully<br>► Overall reduction in total CPU and maximum elapsed time that is maintained<br>► Flat CPU cost 1-up through 8-up<br>► No discernible difference to 8 or 16 way partitioning |
| 6 | ► Updated DB2 table is partitioned 8 ways<br>► All other DB2 tables are non-partitioned<br>► COMMIT processing<br>► Updated DB2 table space is defined as PGSTEAL=NONE | ► Scales to 32-up successfully<br>► Overall reduction in total CPU and maximum elapsed time that is maintained<br>► Flat CPU cost 1-up through 8-up<br>► No discernible difference to partitioning |
| 7 | ► Updated DB2 table is partitioned 8 ways<br>► All other DB2 tables are non-partitioned<br>► COMMIT processing<br>► Updated DB2 table space is defined as PGSTEAL=NONE<br>► Batch SPUFI runs to load object into buffer pool | ► Scales to 32-up successfully<br>► Overall reduction in total CPU and maximum elapsed time that is maintained<br>► Flat CPU cost 1-up through 8-up<br>► No discernible difference to test 6 |
| 8 | ► Updated DB2 table is partitioned 8 ways<br>► All other DB2 tables are non-partitioned<br>► COMMIT processing<br>► Updated DB2 table space is defined as PGSTEAL=NONE<br>► Batch SPUFI runs to load object into buffer pool<br>► Increased buffer pool size for in memory object | ► Scales to 32-up successfully<br>► Overall reduction in total CPU and maximum elapsed time that is maintained<br>► Flat CPU cost 1-up through 8-up<br>► No discernible difference to test 7 |

In this publication, all timings are in minutes. In the rest of this section, we have graphs that show the following items:

| | |
|---|---|
| **Max elapsed** | The elapsed time of the longest-running instance |
| **Total CPU** | The sum of the CPU time for all instances |

The “1-up” data point refers to the program that is modified, for example, by removing reporting, you prepare to run more than one instance. The “0-up” data point refers to the original unmodified program.

Figure 7-9 contains a graph showing the total CPU used by each of the test runs.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| Run 1 | 19.42 | 19.39 | | | | | |
| Run 2 | 19.41 | 19.22 | 19.84 | 20.24 | 20.76 | | |
| Run 3 | 11.22 | 11.22 | 11.22 | 11.20 | 11.20 | 15.40 | 13.41 |
| Run 4 | 11.23 | 11.23 | 11.22 | 11.20 | 11.20 | 15.40 | 13.30 |
| Run 5 | 11.23 | 11.25 | 11.23 | 11.22 | 11.20 | 15.40 | 13.30 |
| Run 6 | 11.22 | 11.22 | 11.22 | 11.21 | 11.20 | 15.40 | 13.30 |
| Run 7 | 11.22 | 11.22 | 11.22 | 11.20 | 11.20 | 15.40 | 13.30 |
| Run 8 | 11.22 | 11.22 | 11.22 | 11.20 | 11.20 | 15.40 | 13.30 |

*Figure 7-9 Total CPU used by each of the test runs*

Runs 1 and 2 show the worst CPU picture, with the CPU usage essentially the same between them. Although Run 2 is modified to commit frequently enough to allow scaling beyond 1-way, it is not changed radically enough to reduce CPU. The total CPU degrades slightly as the number of clones increases.

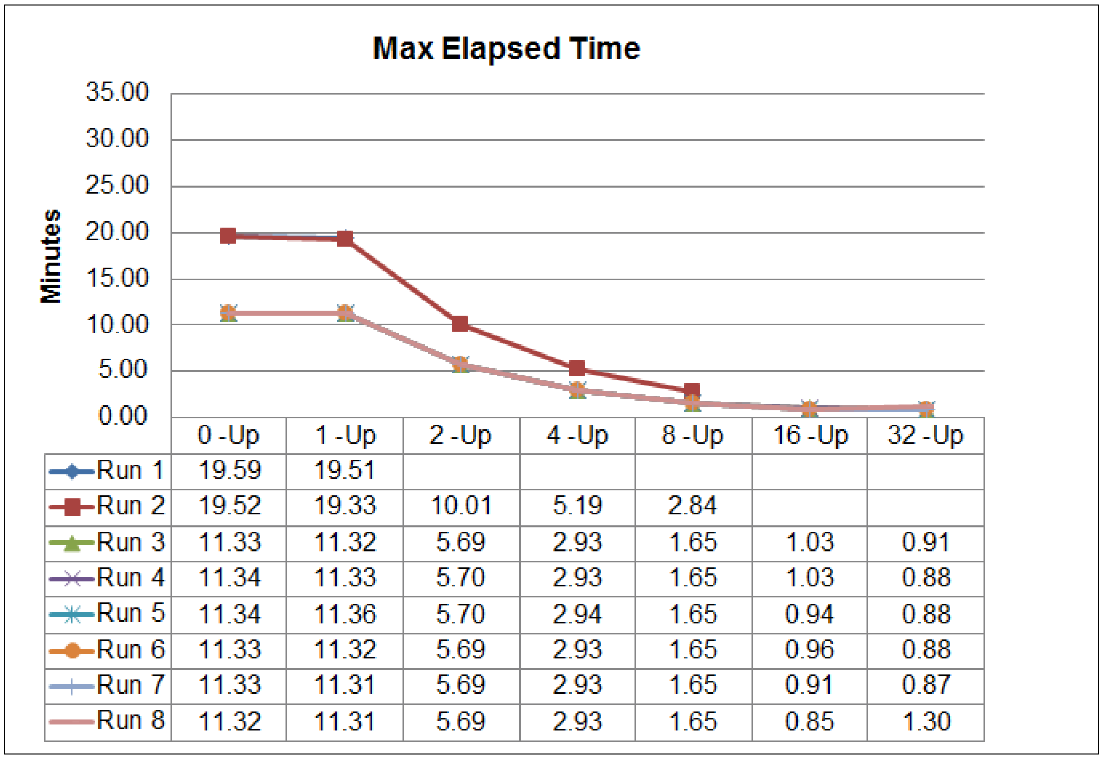
Run 3 is modified to partition the updated table eight ways. Because each clone updates just a single partition rather than the entire table, the CPU drops dramatically. The Run 3 reduction is repeated across all subsequent runs, with the CPU picture remaining unchanged between them.

The increase in CPU between 8-up and 16-up remains a mystery, particularly as the CPU drops again for the 32-up case. This dynamic is repeated across all the runs, so it is a real effect.

Increasing the number of partitions for the updated table to 16 ways (Run 4) and 32 ways (Run 5) has no noticeable effect on CPU.

Runs 6, 7 and 8 are experiments with buffering. None of them affect CPU time.

Figure 7-10 on page 89 contains a graph showing the maximum elapsed time for each of the test runs.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| Run 1 | 19.59 | 19.51 | | | | | |
| Run 2 | 19.52 | 19.33 | 10.01 | 5.19 | 2.84 | | |
| Run 3 | 11.33 | 11.32 | 5.69 | 2.93 | 1.65 | 1.03 | 0.91 |
| Run 4 | 11.34 | 11.33 | 5.70 | 2.93 | 1.65 | 1.03 | 0.88 |
| Run 5 | 11.34 | 11.36 | 5.70 | 2.94 | 1.65 | 0.94 | 0.88 |
| Run 6 | 11.33 | 11.32 | 5.69 | 2.93 | 1.65 | 0.96 | 0.88 |
| Run 7 | 11.33 | 11.31 | 5.69 | 2.93 | 1.65 | 0.91 | 0.87 |
| Run 8 | 11.32 | 11.31 | 5.69 | 2.93 | 1.65 | 0.85 | 1.30 |

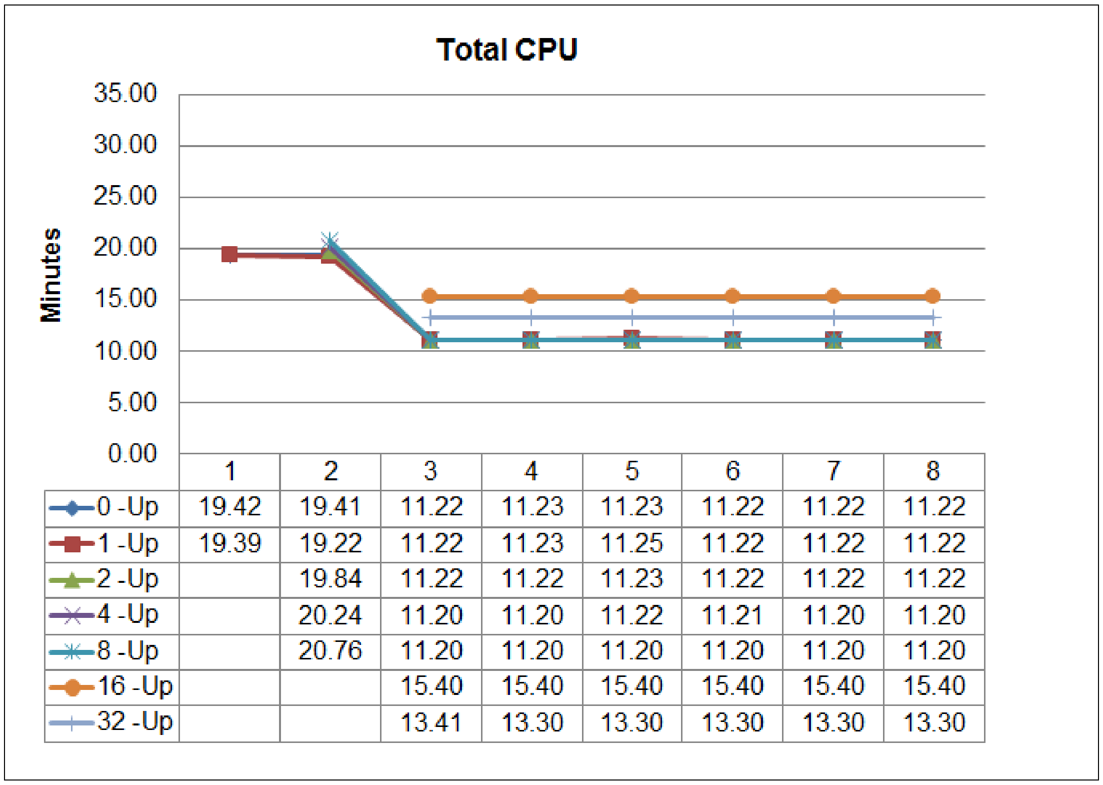
*Figure 7-10 Maximum elapsed time for each of the test runs*

Runs 1 and 2 (before partitioning the updated table) show identical maximum elapsed times, but Run 2 shows COMMIT processing, which allows the application to scale up to 8-up with increasing benefit of more clones.

Runs 3 onwards show the benefit of partitioning the update table similar to the CPU reductions. We scale with increasing benefit up to 32-up. As the benefit of going from 16-up to 32-up is only slight, it is reasonable to conclude going beyond 32-up does not provide a significant benefit.

Runs 4 onwards show no further increases in speed.

Figure 7-11 is a graph showing the total CPU that is used by the number of clones.

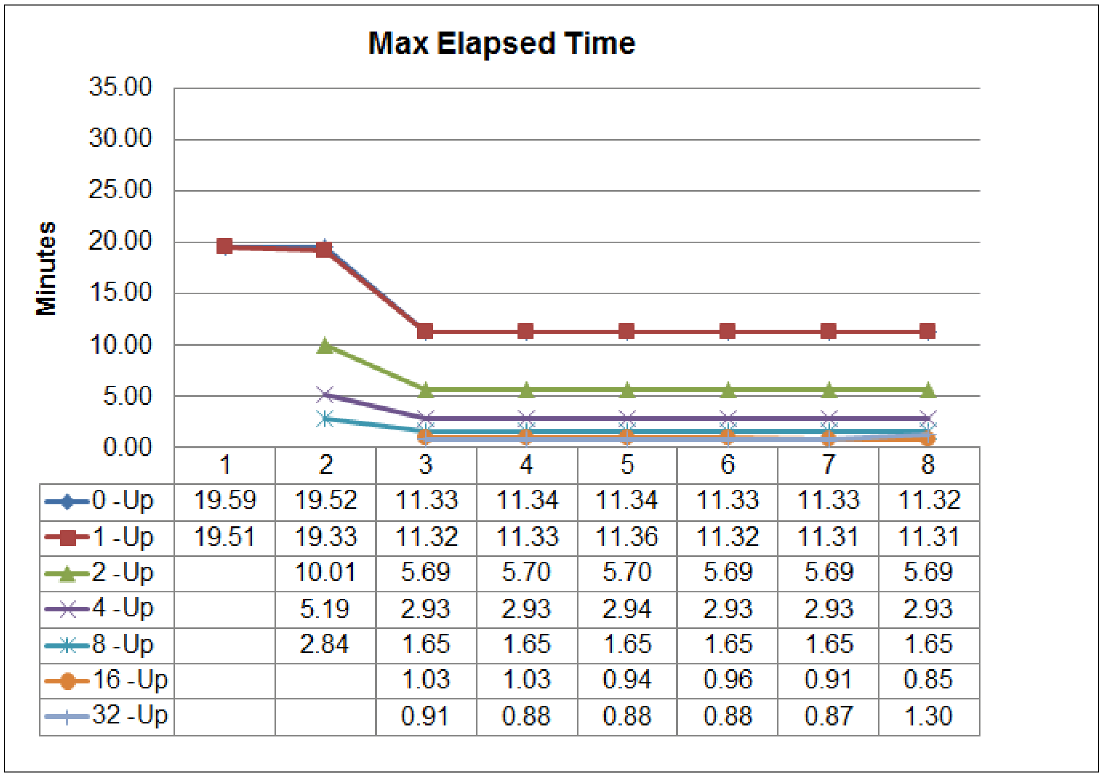


| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 -Up | 19.42 | 19.41 | 11.22 | 11.23 | 11.23 | 11.22 | 11.22 | 11.22 |
| 1 -Up | 19.39 | 19.22 | 11.22 | 11.23 | 11.25 | 11.22 | 11.22 | 11.22 |
| 2 -Up | | 19.84 | 11.22 | 11.22 | 11.23 | 11.22 | 11.22 | 11.22 |
| 4 -Up | | 20.24 | 11.20 | 11.20 | 11.22 | 11.21 | 11.20 | 11.20 |
| 8 -Up | | 20.76 | 11.20 | 11.20 | 11.20 | 11.20 | 11.20 | 11.20 |
| 16 -Up | | | 15.40 | 15.40 | 15.40 | 15.40 | 15.40 | 15.40 |
| 32 -Up | | | 13.41 | 13.30 | 13.30 | 13.30 | 13.30 | 13.30 |

*Figure 7-11 Total CPU that is used by the number of clones*

The difference in the total CPU that is consumed between 0-up, 1-up, 2-up, 4-up, and 8-up is insignificant. Only when there are 16 clones or 32 does the CPU increase. In this test case, the most important difference is not in the number of clones but between Run 2 and Run 3.

Figure 7-12 on page 91 is a graph that shows the maximum elapsed time that is taken by the number of clones.



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 -Up | 19.59 | 19.52 | 11.33 | 11.34 | 11.34 | 11.33 | 11.33 | 11.32 |
| 1 -Up | 19.51 | 19.33 | 11.32 | 11.33 | 11.36 | 11.32 | 11.31 | 11.31 |
| 2 -Up | | 10.01 | 5.69 | 5.70 | 5.70 | 5.69 | 5.69 | 5.69 |
| 4 -Up | | 5.19 | 2.93 | 2.93 | 2.94 | 2.93 | 2.93 | 2.93 |
| 8 -Up | | 2.84 | 1.65 | 1.65 | 1.65 | 1.65 | 1.65 | 1.65 |
| 16 -Up | | | 1.03 | 1.03 | 0.94 | 0.96 | 0.91 | 0.85 |
| 32 -Up | | | 0.91 | 0.88 | 0.88 | 0.88 | 0.87 | 1.30 |

*Figure 7-12 Maximum elapsed time that is taken by the number of clones*

The elapsed time is most affected by the number of clones. This is exactly what we want because we are cloning to reduce elapsed time.

In this test case, the early runs, which each represents a modification to the application, provide the basis for scalability. We cannot run any clones without the modification that is introduced in Run 2 (adding commit logic). The modification in Run 3 (partitioning the updated table) enables us to run more than 8-up.

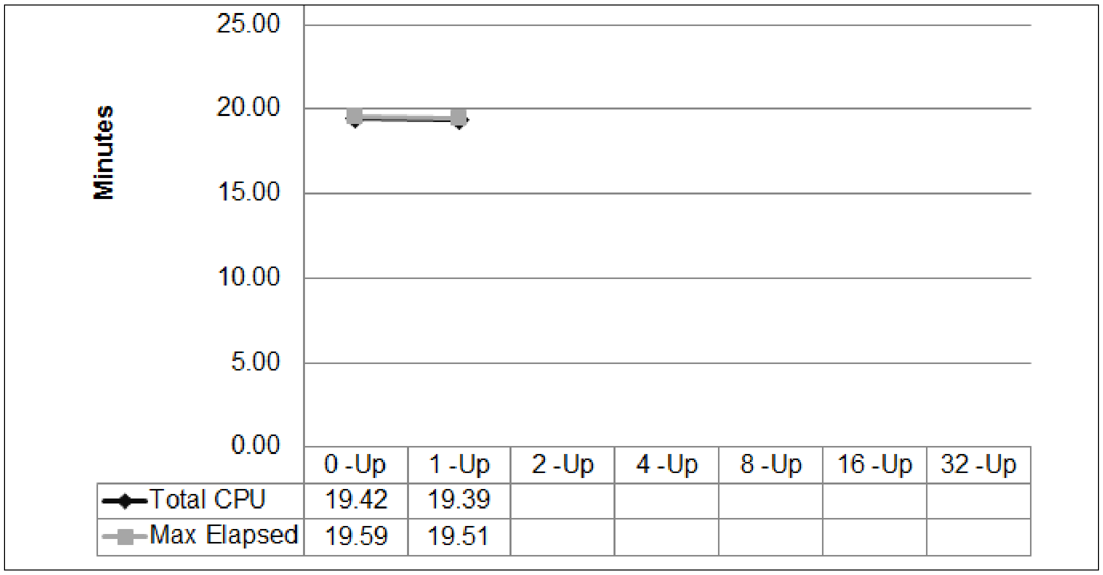
Depending on business objectives, if this were a real case, you might regard the 8-up case as the best trade-off between elapsed time reduction and CPU consumed. If elapsed time is an overriding consideration, 16-up or 32-up cloning might be preferable.

## 7.2.1 Base case

In this first test, the DB2 tables are in table spaces containing a single table and none of them are partitioned. The programs have no DB2 commit processing.

### Results

The results from the first test run are shown in Figure 7-13.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| ◆ Total CPU | 19.42 | 19.39 | | | | | |
| ■ Max Elapsed | 19.59 | 19.51 | | | | | |

*Figure 7-13 CPU and elapsed time - no commits*

Without commit processing, it is only possible to run a single instance of the code.

In the 2-up test, one instance times out while waiting for the other instance to release the locks on a DB2 table space page. DB2 returns an **`SQLCODE -911`** with **`SQLSTATE 40001`** and **`Reason Code 00C9008E`** on resource **`TYPE 00000302`**.

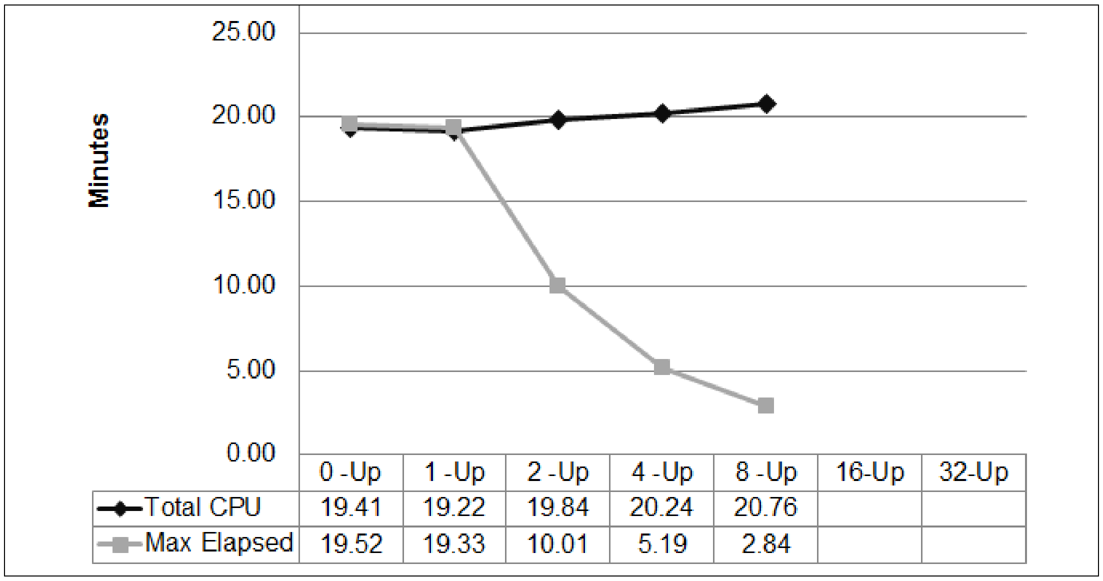
DB2 Accounting Trace shows all the time that is spent in CPU. About half the time is spent in DB2 and half in application code.

## 7.2.2 First refinement: DB2 COMMIT processing

The only change that is made for this test run is to add DB2 COMMIT processing to POKB303. A COMMIT is run after every UPDATE.

### Results

The results from the test run with DB2 COMMITs are shown in Figure 7-14 on page 93.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16-Up | 32-Up |
|---|---|---|---|---|---|---|---|
| ◆ Total CPU | 19.41 | 19.22 | 19.84 | 20.24 | 20.76 | | |
| ■ Max Elapsed | 19.52 | 19.33 | 10.01 | 5.19 | 2.84 | | |

*Figure 7-14 CPU and elapsed time with DB2 commit processing*

In our example, we achieve successful tests from 0-up through 8-up.

There is a slight increase in CPU as the number of instances increase and a decrease in the maximum elapsed time for any one slice. As with the base case, almost all the time is CPU time. Non DB2 CPU is slightly higher than DB2 CPU across all data points.

Other DB2 time components are present in small quantities: Each run has a few seconds of lock/latch wait, and DB2 services extend/delete/define, or read asynchronous wait time. None of these components are significant enough to warrant tuning.

In the 16-up test, the third instance failed with a deadlock situation with the fourth instance on a table space page. DB2 returns an **`SQLCODE -911`** with **`SQLSTATE 40001`** and **`Reason Code 00C90088`** on resource **`TYPE 00000302`**.

By issuing a DB2 COMMIT after every update, the implementation scales much higher than the base case. Running eight instances achieves a less-than-3 minute run time, compared to almost 20 minutes, for a small increase in CPU. Assuming the application runs reliably with eight instances, 8-up is the preferred configuration.

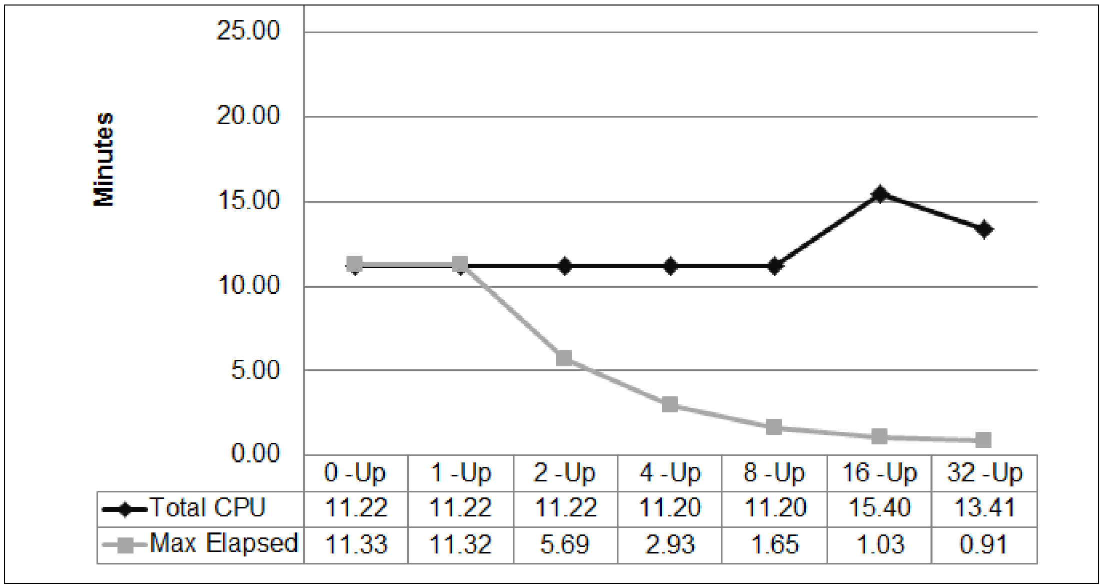
## 7.2.3 Second refinement: Partitioning the updated DB2 table

The change that is made for this test run is to partition one of the DB2 tables. The table space containing the DB2 table that is updated by the programs is altered to a partition-by-range universal table space with eight partitions.

The table partitioning is defined on the column that corresponds to the data in the driving QSAM file. This matches how the input file is being divided in the split process. Each of the partitions is defined to contain as equal a volume of data as possible.

### Results

The results from the test are shown in Figure 7-15.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| Total CPU | 11.22 | 11.22 | 11.22 | 11.20 | 11.20 | 15.40 | 13.41 |
| Max Elapsed | 11.33 | 11.32 | 5.69 | 2.93 | 1.65 | 1.03 | 0.91 |

*Figure 7-15 CPU and elapsed time with eight partitions for the updated table*

In this test, we achieve successful tests from 0-up through to 32-up with no application issues. Both the CPU and elapsed times are almost halved. There is almost no impact to the CPU usage as the number of instances increases. The healthy speed improvements are maintained across all levels of splitting.

Almost all the in DB2 CPU disappears as partitioning leads to a much reduced level of data access. The non DB2 CPU remains at the same level as previous iterations.

At higher levels of parallelism, CPU queuing becomes an important factor. This is particularly so in the 32-up case, where about half the run time is CPU queuing.

In the 8-up case, a small amount of lock/latch wait time is experienced.

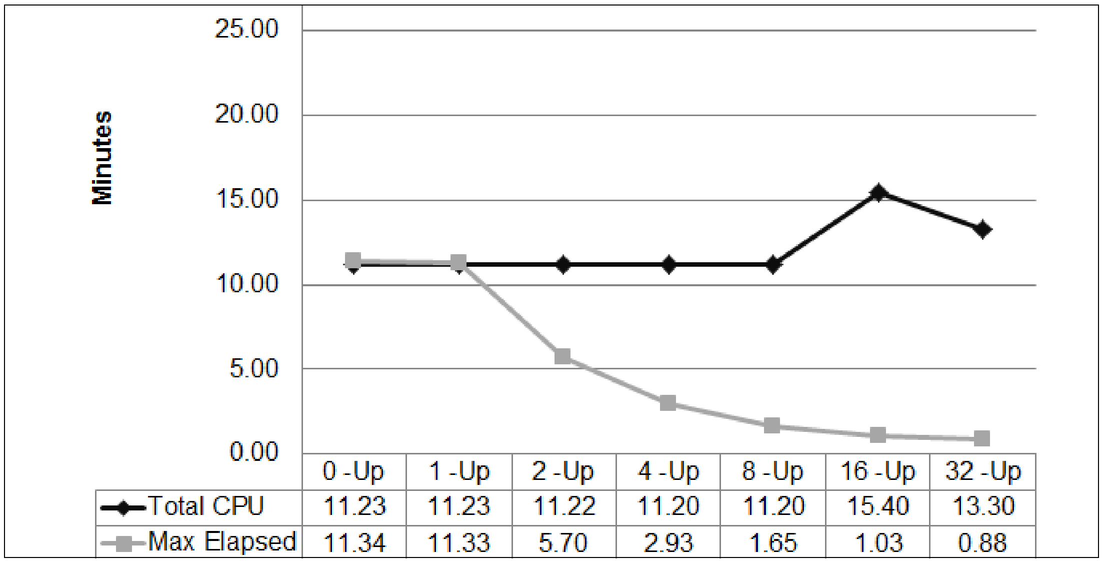
With partitioning, the run time is now around 1 minute, compared to the original 20 minutes. Although the 8-up or 16-up configurations are probably the best, the limiting factor is now CPU provisioning. If more CPU is provisioned, the application probably runs in around half a minute.

## 7.2.4 Third refinement: Increasing the number of partitions

In this test run, the number of partitions for the updated table is doubled from eight to 16. Again, care is taken to ensure that the volume of data is evenly distributed across the partitions.

### Results

The results from the test are shown in Figure 7-16 on page 95.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| Total CPU | 11.23 | 11.23 | 11.22 | 11.20 | 11.20 | 15.40 | 13.30 |
| Max Elapsed | 11.34 | 11.33 | 5.70 | 2.93 | 1.65 | 1.03 | 0.88 |

*Figure 7-16 CPU and elapsed time with 16 partitions for the updated table*

In this test, doubling the number of partitions makes no significant difference to either the CPU consumed or to the maximum elapsed time. The same is true of doubling the number of partitions again to 32.

Again, most of the time is non DB2 CPU time, with CPU queuing being more than half the elapsed time in the 32-up case. In the 16-up case, some lock/latch wait time is evident.

For our example, the most effective number of partitions is eight, which corresponds to 8-up and 16-up configurations giving a nice increase in speed.

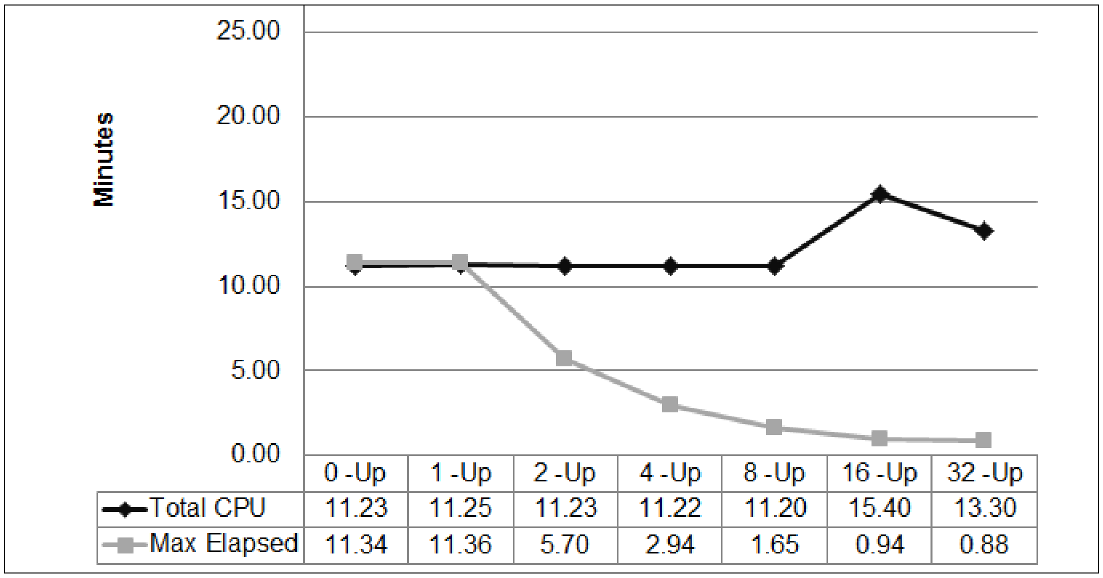
## 7.2.5 Fourth refinement: Aligning partitions to input

In this test run, the number of partitions for the updated table is increased to 32. Instead of distributing the data evenly by volume across the partitions, the partition boundaries deliberately are aligned to correspond with the data in our input file. Each instance accesses and updates data in a single partition with no overlaps.

This technique might not always be a practical solution, but can be useful for applications where each instance processes data for a sensible partitioning key, for example, a time period, or physical or logical locations.

### Results

The results from the test are shown in Figure 7-17.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| ◆ Total CPU | 11.23 | 11.25 | 11.23 | 11.22 | 11.20 | 15.40 | 13.30 |
| ■ Max Elapsed | 11.34 | 11.36 | 5.70 | 2.94 | 1.65 | 0.94 | 0.88 |

*Figure 7-17 CPU and elapsed time with 32 partitions for the updated table*

In our example, aligning the partitioned data to the input file makes no significant difference to either the CPU cycles that are consumed or to the maximum elapsed time. Analysis of DB2 Accounting Trace shows that the detailed elapsed time components also are unchanged.

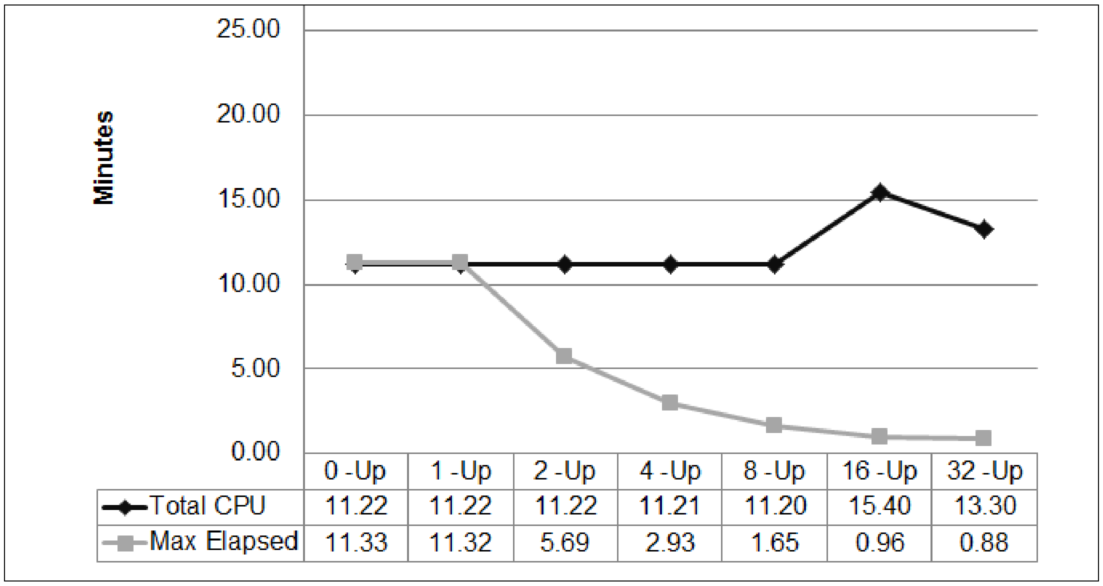
## 7.2.6 Fifth refinement: In-memory table space

In this test run, the number of partitions for the updated table is set back to eight.

A feature that was introduced in DB2 10 is the ability to define a table space as being in memory. This is done by setting `PGSTEAL=NONE` on a buffer pool and then creating the table space in that buffer pool. This enables you to use large buffer pools to preinstall objects into the buffer pool on first reference, which can improve the I/O performance.

### Results

The results from the test are shown in Figure 7-18 on page 97.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| ◆ Total CPU | 11.22 | 11.22 | 11.22 | 11.21 | 11.20 | 15.40 | 13.30 |
| ■ Max Elapsed | 11.33 | 11.32 | 5.69 | 2.93 | 1.65 | 0.96 | 0.88 |

*Figure 7-18 CPU and elapsed time with 8 partitions and in memory table space for the updated table*

The CPU and elapsed times are again unchanged. Analysis of DB2 Accounting Trace shows that the detailed elapsed time components are also unchanged. Accounting Trace from the run that is described in 7.2.3, “Second refinement: Partitioning the updated DB2 table” on page 93 explains why this technique did not help: There was little elapsed time in the components that are related to buffer pools. These components are synchronous database wait, asynchronous read wait, and asynchronous write wait.

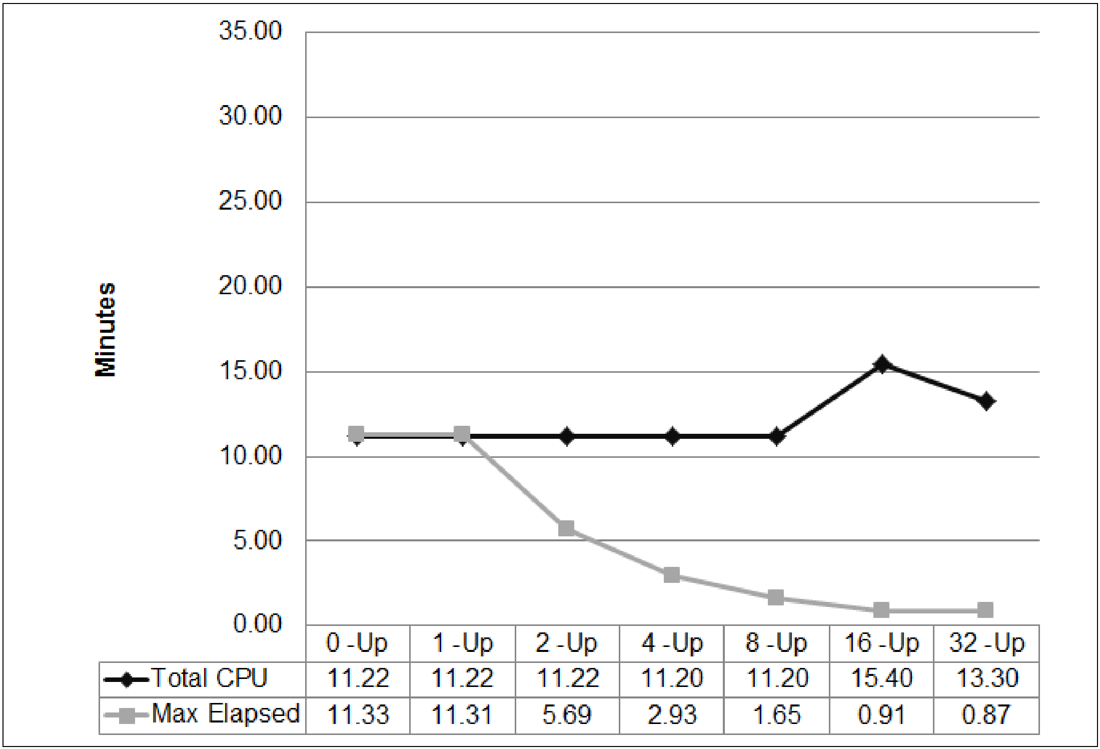
## 7.2.7 Sixth refinement: In-memory table space with first reference

This test is the same as the fifth change point, apart from the addition of a step after DB2 is started. A batch SPUFI is run to access the table that is in the in memory table space. In our case, this is the table that the programs update. This is done so that the SPUFI, and not our test program, is the “first reference” on the table, which prompts DB2 to load the object into the buffer pool.

The CPU time and the elapsed time of the SPUFI are not included in the results.

## Results

The results from the test are shown in Figure 7-19.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| ◆ Total CPU | 11.22 | 11.22 | 11.22 | 11.20 | 11.20 | 15.40 | 13.30 |
| ■ Max Elapsed | 11.33 | 11.31 | 5.69 | 2.93 | 1.65 | 0.91 | 0.87 |

*Figure 7-19 Adding a previous "first reference" for the updated table*

To the previous run showing CPU and elapsed time with eight partitions and in-memory table space, we add a previous "first reference" for the updated table.

The CPU and elapsed times are again unchanged. Analysis of DB2 Accounting Trace shows that the detailed elapsed time components are also unchanged. Accounting Trace from the run that is described in 7.2.3, "Second refinement: Partitioning the updated DB2 table" on page 93 explains why this technique did not help: There is little elapsed time in the components that are related to buffer pools. These components are synchronous database wait, asynchronous read wait, and asynchronous write wait.

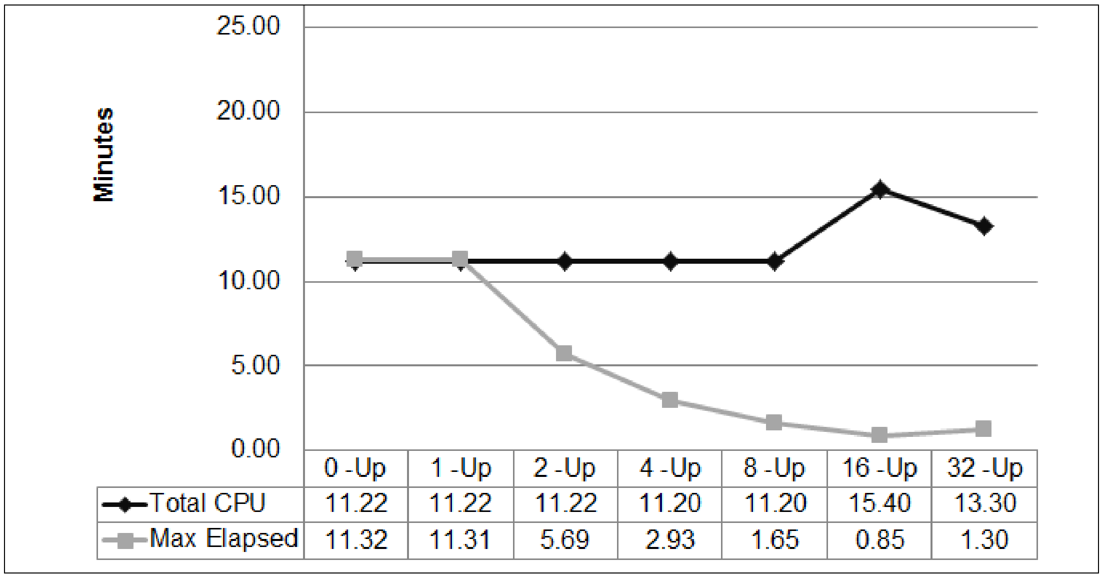
## 7.2.8 Seventh refinement: In-memory table space and increased buffer pool size

This test is the same as the fifth change point because it is attempting to reduce the need to retrieve data from disk by trying to make sure that it is present in a buffer pool. However, not all installations are prepared to define whole table spaces in memory. This alternative option increases the size of the buffer pool itself.

Our prior refinements ran against a buffer pool that is defined with a VPSIZE of 20000. For this test, an `ALTER BUFFERPOOL` command is issued to increase the VPSIZE to 200000.

### Results

The results from the test are shown in Figure 7-20 on page 99.



| | 0 -Up | 1 -Up | 2 -Up | 4 -Up | 8 -Up | 16 -Up | 32 -Up |
|---|---|---|---|---|---|---|---|
| Total CPU | 11.22 | 11.22 | 11.22 | 11.20 | 11.20 | 15.40 | 13.30 |
| Max Elapsed | 11.32 | 11.31 | 5.69 | 2.93 | 1.65 | 0.85 | 1.30 |

*Figure 7-20 Increasing the buffer pool size*

To the fifth run showing CPU and elapsed time with eight partitions and in-memory table space for the updated table, we add an increased buffer pool size.

The CPU and elapsed times are again unchanged. Analysis of DB2 Accounting Trace shows that the detailed elapsed time components are also unchanged. Accounting Trace from the run that is described in 7.2.3, “Second refinement: Partitioning the updated DB2 table” on page 93 explains why this technique did not help: There was little elapsed time in the components that are related to buffer pools. These components are synchronous database wait, asynchronous read wait, and asynchronous write wait.

## 7.3 Relationship to customer installations

The test case initially (the “0-up” data points) consisted of a single job. Real installations run hundreds to tens of thousands of jobs a night, with typical applications composed of dozens of jobs. Even the splits (for example, the “32-up” data points) do not represent reality, as the large inventory of jobs in a typical application are different from each other.

The case study does not work with an existing customer batch job. Instead, we built a job that we believe to be similar to many that are present in customer installations. The objective of the case study and this publication is to guide installations in how to split jobs and what sorts of issues to address.

Although we split using powers of two, this is not the only way to split jobs:

- Installations often have jobs split using a different scheme.
- Splitting by doubling is not necessarily the best way to split a job.

We choose to use powers of two because it rapidly (with few data points) gets us to high levels of splitting. High levels of splitting are useful to us because they expose behaviors that emerge at higher levels of scaling. We also do not try to replicate a customer operating environment. For example, we ran in a CPU-rich and memory-rich LPAR. Although increasing parallelism is known to increase concurrent resource requirements, we do not run experiments to prove that, other than noting that some data points (“32-up” most notably) led to CPU queuing.

A

# Application modernization opportunities

This appendix describes opportunities to modernize batch applications when batch jobs are split. It is an appendix because application modernization is an additional opportunity, not part of the main purpose of this document.

## Repurposing reporting data

When you split a batch job that produces a report, this split creates two types of jobs:

- Report data generation jobs
- A report writing job

The report data generation jobs must create transient data sets to pass to the report writing job. Without splitting the job, only a single transient data set is created. When split, each instance creates a transient data set containing reporting data. A third type of job is required, which is called a *report data merge job*.

The report writing job reads a single data set, whether from a single report data generation job or from the report data merge job.

Consider whether this single data set can be created in a way that other consumers can use. For example:

- The transient data set might be in Extensible Markup Language (XML) format.
- A web server might serve the transient data set as a JavaScript Object Notation (JSON) file.

Neither format is difficult to create. Both formats are consumable by using modern programming techniques. For example:

- Most modern programming languages can process XML. Processing by the z/OS System XML Parser can be offloaded to a zIIP specialty engine.
- Some modern processing languages can process JSON. A good example is JavaScript, whether directly or by using a JavaScript framework, such as Dojo or jQuery.

### Creating an XML file

Here is a sample XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<inventory>
  <item name="ALPHA   ">ONE     </item>
  <item name="BRAVO   ">TWO     </item>
  <item name="CHARLIE ">THREE   </item>
  <item name="DELTA   ">FOUR    </item>
</inventory>
```

Here is the JCL to start **DFSORT** to create it:

```
//MAKEXML  EXEC PGM=ICEMAN
//SYSOUT   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYMNOUT  DD SYSOUT=*
//SYMNAMES DD *
POSITION,1
NAME,*,8,CH
SKIP,1
NUMBER,*,8,CH
/*
//SORTIN   DD *
ALPHA    ONE
BRAVO    TWO
```

```
CHARLIE  THREE
DELTA    FOUR
/*
//SORTOUT  DD SYSOUT=*
//SYSIN    DD *
  OPTION COPY
  OUTFIL FNAMES=SORTOUT,REMOVECC,
  HEADER1=('<?xml version="1.0" encoding="UTF-8"?>',/,
  '<inventory>'),
  OUTREC=(3:C'<item name="',NAME,C'">',NUMBER,C'</item>'),
  TRAILER1=('</inventory>')
  END
/*
```

In this example, the data is in-stream and accessed by the **`SORTIN DD`** statement. The data, which is composed of two fields, is mapped by using **`DFSORT`** Symbols. The Symbols are defined by the instream data set that is pointed to by the **`SYMNAMES DD`** statement.

The **`OUTREC`** clause formats each input line. Trailing blanks in the input fields are retained because they are significant in XML, although it is possible to remove them.

The `HEADER1` and `TRAILER1` clauses describe the static XML header and trailer lines.

## Creating a JSON file

JSON files can be created in a similar way to XML files. This example is derived from "Creating an XML file" on page 102. Here is the JSON file:

```
{
"inventory": [
  {"name": "ALPHA   ","number": "ONE     "}
  ,{"name": "BRAVO   ","number": "TWO     "}
  ,{"name": "CHARLIE ","number": "THREE   "}
  ,{"name": "DELTA   ","number": "FOUR    "}
]
}
```

There is an additional complication. Although there are no separators between nodes in XML, JSON requires commas between items, including between elements of arrays. The complication arises because there must not be a comma before the first item or after the last item. The following JCL is more complex than in the XML example because you must carefully control when commas are inserted:

```
//MAKEJSON EXEC PGM=ICEMAN
//SYSOUT   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYMNOUT  DD SYSOUT=*
//SYMNAMES DD *
POSITION,1
NAME,*,8,CH
SKIP,1
NUMBER,*,8,CH
/*
//SORTIN   DD *
ALPHA    ONE
BRAVO    TWO
```

```
CHARLIE  THREE
DELTA    FOUR
/*
//SORTOUT  DD SYSOUT=*
//SYSIN    DD *
  OPTION COPY
*
  INREC IFTHEN=(WHEN=INIT,BUILD=(SEQNUM,4,BI,
          C'{"name": "',NAME,C'","number": "',NUMBER,C'"}')),
        IFTHEN=(WHEN=(1,4,BI,GT,+1),BUILD=(2X,C',',5,70)),
        IFTHEN=(WHEN=(1,4,BI,EQ,+1),BUILD=(2X,5,70))
*
  OUTFIL FNAMES=SORTOUT,REMOVECC,
  HEADER1=('{'/,
  '"inventory": ['),
  TRAILER1=(']',/,
  '}')
  END
/*
```

If you pass this JSON file through a web-based validator, such as JSON Lint (`http://jsonlint.com`), the file is reported as clean JSON. JSON Lint also reformats it, although the JSON Lint version is not more parseable.

This JCL sample is more complex than the previous example. All the additional complexity is in the **INREC** statement. (This processing might be performed in an **OUTREC** or even an **OUTFIL OUTREC** statement.)

The **INREC** statement has three `IFTHEN` clauses (or stages):

1. Always processed, producing the formatted line with a 4-byte sequence number on the front. The sequence number starts at 1 and is in binary format.
2. Processed if the sequence number is greater than 1, placing a comma (and two indenting spaces) in front of the formatted line.
3. Processed if the sequence number is 1, placing the two spaces (and no comma) in front of the formatted line.

Using `IFTHEN` stages in this way is the standard “treat each record according to its characteristics and through multiple stages” approach you can take with **DFSORT IFTHEN**.

This technique produces one line without a comma and the following ones with a comma, satisfying JSON syntax rules.

## Trimming trailing blanks in the output file

Consider the following space-saving refinement. If you want to create output in variable-length record format (`RECFM=VB`) without trailing spaces, add the following code to the **OUTFIL** statement:

```
FTOV,VLTRIM=C' '
```

Also, code the output data set DD to specify variable-length record format.

## Modernizing report generation

Many existing reports are old. For example, line-mode formats (such as 1403 format) were widespread in the 1960s through the 1980s.

Consider using the separation of report data generation and report writing to modernize the report.

Modern consumers of reports often prefer formats such as the Adobe Portable Document Format (PDF), which has the following advantages:

- More beautiful and flexible report formatting capabilities, such as fonts and graphics.
- The ability to serve documents through modern infrastructure, such as web servers. Most modern web browsers can display PDF documents that are served by a web server.

One technique to consider is processing generated XML, as described in "Repurposing reporting data" on page 102. A standard method is to use the XML Extensible Stylesheet Language Transformations (XSLT) Language and its document generation counterpart, XSL Formatting Objects (XSL-FO), to create documents.

B

# Operational supplement

This appendix provides supplementary information for Chapter 5, “Operational considerations” on page 29. It is not necessary for a high-level understanding of the material, but it provides follow-on reading, routes to additional materials, and code samples that can be used as a basis for experimentation or solutions.

Any code that is included in and referred to from this appendix is provided *as is* with no additional support.

## Obtaining sample exits and the SOE toolkit

Some examples in Chapter 5, “Operational considerations” on page 29 use an assembly language implementation of the Job-Library-Read exit (EQQUX002) and a REXX toolkit for the IBM Tivoli Workload Scheduler for z/OS Program Interface (PIF) known as the Scheduling Operational Environment (SOE). Both tools can be downloaded from the IBM developerWorks z/Glue community. At present, they are not formally part of the product. As such, they are supported only by the community at the z/Glue website that is found at:

https://www.ibm.com/developerworks/community/groups/community/zGlue

## REXX utility JCLCATDD

The ISPF/REXX utility JCLCATDD ha is written to provide a way of building a **DD** statement containing data sets that meet certain naming criteria. It is included in the forthcoming release of the SOE toolkit (Version 3.3), but the source code is included here so that you can use the techniques and assess whether they should be included in your own code.

The routine allows several keywords to be passed to it as arguments to influence the contents of the JCL statement. The arguments are passed in the form KEYWORD(VALUE) through the **SYSIN DD** statement. The output is written to a file that is allocated to OUTJCL.

Here are the valid keywords:

- **DDNAME**: The JCL label for the concatenation.
- **DISP**: Disposition for all the data sets in the concatenation.
- **DSLIST**: List building argument. This can be one or more data set masks that are separated by commas. The mask syntax is the same that is used by PDF option 3.4.
- **DSLIM**: The minimum and maximum number of data sets that are expected, separated by commas. If nothing is specified, a minimum of one is expected with no maximum. If the limits are breached, the REXX ends with return code 8.

The source code is shown in Figure B-1 on page 109, Figure B-2 on page 110, Figure B-3 on page 111, and Figure B-4 on page 112.

```
/* REXX */
/*------------------------------------------------------------------+
 | JCLCATDD - Create JCL Concatenation INCLUDE member               |
 |                                                                  |
 | DDNAME   - Label for the concatenation                           |
 | DISP     - Disposition for datasets                              |
 | DSLIST   - List building argument                                |
 | DSLIM    - Lower and upper list limits                           |
 +------------------------------------------------------------------*/
/*------------------------------------------------------------------+
 | Ensure failures set ZISPFRC to 12 (reset by controlled end of pgm) |
 +------------------------------------------------------------------*/
ZISPFRC = 12
ISPEXEC "VPUT (ZISPFRC) SHARED"
ISPEXEC "CONTROL ERRORS RETURN"

/*------------------------------------------------------------------+
 | Read the SYSIN                                                   |
 +------------------------------------------------------------------*/
"EXECIO * DISKR SYSIN (STEM sysin. FINIS)"
IF RC <> 0 THEN DO
   RETURN ERROR("Unable to read SYSIN")
END

/*------------------------------------------------------------------+
 | Bring SYSIN into 1 line (dropping line number area just in case)  |
 +------------------------------------------------------------------*/
in_PARMS = ""
DO i_Row = 1 TO sysin.0
   in_PARMS = STRIP(in_PARMS LEFT(sysin.i_Row,72))
END

/*------------------------------------------------------------------+
 | Initialize variables                                             |
 +------------------------------------------------------------------*/
in.0 = "DDNAME DISP DSLIST DSLIM"
in.DDNAME = ""
in.DISP   = "SHR"
in.DSLIST = ""
in.DSLIM  = "1"
```

*Figure B-1 JCLCATDD REXX source - part 1*

```
/*-------------------------------------------------------------------+
 | Parse out basic keywords                                          |
 +-------------------------------------------------------------------*/
in_PARMS = TRANSLATE(in_PARMS," ",",")

DO WHILE in_PARMS <> ""
   PARSE UPPER VAR in_PARMS in_Key "(" in_Value ")" in_PARMS
   in_PARMS  = STRIP(in_PARMS)
   in_Key    = STRIP(in_Key)
   in_Value  = TRANSLATE(in_Value,"  ",",'")
   in_Value  = STRIP(SPACE(TRANSLATE(in_Value,' ','"'),1))
   in.in_Key = in_Value
   IF WORDPOS(in_Key,in.0) = 0 THEN DO
      RETURN ERROR("Unrecognized keyword" in_Key)
   END
END

/*-------------------------------------------------------------------+
 | Exit if no DDNAME is specified                                    |
 +-------------------------------------------------------------------*/
IF in_DDNAME = "" THEN DO
   RETURN ERROR("No DDNAME specified")
END

/*-------------------------------------------------------------------+
 | Establish count limits                                            |
 +-------------------------------------------------------------------*/
SELECT
   WHEN WORDS(in.DSLIM) = 1 THEN DO
      ds_Min = in.DSLIM
      ds_Max = ""
      IF DATATYPE(ds_Min,"W") = 0 THEN DO
         RETURN ERROR("Minimum limit invalid" ds_Min)
      END
   END
   WHEN WORDS(in.DSLIM) = 2 THEN DO
      PARSE VAR in.DSLIM ds_Min ds_Max
      IF DATATYPE(ds_Min,"W") = 0 THEN DO
         RETURN ERROR("Minimum limit invalid" ds_Min)
      END
      IF DATATYPE(ds_Max,"W") = 0 THEN DO
         RETURN ERROR("Maximum limit invalid" ds_Max)
      END
   END
   OTHERWISE DO
      RETURN ERROR("Invalid limit specification" in.DSLIM)
   END
END
```

*Figure B-2 JCLCATDD REXX source - part 2*

```
/*-------------------------------------------------------------------+
 | Check that there are dataset masks in the list                   |
 +-------------------------------------------------------------------*/
ds_Masks = WORDS(in.DSLIST)
IF ds_Masks = 0 THEN DO
   RETURN ERROR("No DSLIST specified")
END

/*-------------------------------------------------------------------+
 | Find the matching datasets                                       |
 +-------------------------------------------------------------------*/
dd_DSNUM = 0
dd_DSN.  = ""
DO ds_Count = 1 TO ds_Masks
   ds_List = WORD(in.DSLIST,ds_Count)
   ISPEXEC "LMDINIT LISTID(DSID) LEVEL("||ds_List||")"
   DSOUT = ""
   DO FOREVER
      ISPEXEC "LMDLIST LISTID("||DSID||") OPTION(LIST)" ,
              "DATASET(DSOUT)"
      IF RC > 0 THEN LEAVE
      dd_DSNUM = dd_DSNUM + 1
      dd_DSN.dd_DSNUM = DSOUT
   END
   ISPEXEC "LMDLIST LISTID("||DSID||") OPTION(FREE)"
   ISPEXEC "LMDFREE LISTID("||DSID||")"
END

/*-------------------------------------------------------------------+
 | Find the matching datasets                                       |
 +-------------------------------------------------------------------*/
jcl_Out = 0
SELECT
   WHEN (ds_Min = 0) & (dd_DSNUM = 0) THEN DO
      jcl_Out = 1
      jcl.jcl_Out = LEFT("//"||in.DDNAME,10) "DD DUMMY"
   END
   WHEN (dd_DSNUM < ds_Min) THEN DO
      RETURN ERROR("Too few datasets found:" dd_DSNUM ,
                   "(Min="||ds_Min||")")
   END
   WHEN (dd_DSNUM > ds_Max) & (ds_Max <> "") THEN DO
      RETURN ERROR("Too many datasets found:" dd_DSNUM ,
                   "(Max="||ds_Max||")")
   END
   OTHERWISE NOP
END
```

*Figure B-3 JCLCATDD REXX source - part 3*

```
/*------------------------------------------------------------------+
 | Build the JCL                                                    |
 +------------------------------------------------------------------*/
IF WORDS(in.DISP) > 1 THEN
   in.DISP = "("||TRANSLATE(in.DISP,","," ")||")"
DO i_JCL = 1 TO dd_DSNUM
   IF i_JCL = 1 THEN
      jcl_Base = LEFT("//"||in.DDNAME,10) "DD DISP="||in.DISP||","
   ELSE
      jcl_Base = LEFT("//",10) "DD DISP="||in.DISP||","
   jcl_Dsn = "DSN="||dd_DSN.i_JCL
   IF LENGTH(jcl_Base||jcl_DSN) > 71 THEN DO
      jcl_Out = jcl_Out + 1
      jcl.jcl_Out = jcl_Base
      jcl_Out = jcl_Out + 1
      jcl.jcl_Out = LEFT("//",10) jcl_DSN
   END
   ELSE DO
      jcl_Out = jcl_Out + 1
      jcl.jcl_Out = jcl_Base||jcl_DSN
   END
END

/*------------------------------------------------------------------+
 | Export the JCL                                                   |
 +------------------------------------------------------------------*/
"EXECIO * DISKW OUTJCL (STEM jcl. FINIS)"
IF RC > 0 THEN DO
   RETURN ERROR("Unable to write OUTJCL RC="||RC)
END

/*------------------------------------------------------------------+
 | Exit cleanly                                                     |
 +------------------------------------------------------------------*/
ZISPFRC = 0
ISPEXEC "VPUT (ZISPFRC) SHARED"
RETURN ZISPFRC

/*------------------------------------------------------------------+
 | Error message return                                             |
 +------------------------------------------------------------------*/
ERROR:
   PARSE ARG err_Message
   ZISPFRC = 8
   SAY err_Message
   ISPEXEC "VPUT (ZISPFRC)" SHARED
RETURN ZISPFRC
```

*Figure B-4 JCLCATDD REXX source - part 4*

Running JCL for this REXX utility requires that you create an ISPF environment for it to run in because it uses the ISPF services LMDINIT, LMDLIST, and LMDFREE to build the data set lists.

Figure B-5 shows an example of how to run JCLCATDD. In the example, the REXX is contained within data set BATCH.CLIST and writes the output to SYSOUT. Adjust the data set names to suit your environment.

```
//JCLCATDD EXEC PGM=IKJEFT01,
//         PARM='ISPSTART CMD(JCLCATDD)'
//SYSPROC  DD DISP=SHR,DSN=BATCH.CLIST
//ISPPROF  DD DISP=(NEW,DELETE),DSN=&&ISPPROF,
//         UNIT=SYSDA,SPACE=(TRK,(75,15,10)),
//         RECFM=FB,DSORG=PO,LRECL=80
//ISPLOG   DD SYSOUT=*,RECFM=VA,LRECL=125
//ISPPLIB  DD DISP=SHR,DSN=ISP.SISPPENU
//ISPMLIB  DD DISP=SHR,DSN=ISP.SISPMENU
//ISPSLIB  DD DISP=SHR,DSN=ISP.SISPSENU
//ISPTLIB  DD DISP=SHR,DSN=ISP.SISPTENU
//OUTJCL   DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD DUMMY
//SYSIN    DD *
DDNAME(MERGEDD)
DSLIST(BAKERC.DAL.*)
```

*Figure B-5 Example JCL for running JCLCATDD*

This example produces JCL for a **DD** statement that is called **MERGEDD** showing all data sets beginning with BAKERC.DAL.

# Related publications

The publications that are listed in this section are considered suitable for a more detailed description of the topics that are covered in this paper.

## IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Some publications referenced in this list might be available in softcopy only.

- *Approaches to Optimize Batch Processing on z/OS*, REDP-4816
- *Batch Modernization on z/OS*, SG24-7779
- *DB2 11 for z/OS Technical Overview*, SG24-8180
- *DB2 10 for z/OS Performance Topics*, SG24-7942
- *System/390 MVS Parallel Sysplex Batch Performance*, SG24-2557

You can search for, view, download, or order these documents and other IBM Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

**ibm.com**/redbooks

## Other publications

These publications are also relevant as further information sources:

- *DB2 11 for z/OS Administration Guide,* SC19-4050
- *DB2 11 for z/OS Application Programming and SQL Guide,* SC19-4051
- *DB2 11 for z/OS Managing Performance,* SC19-4060
- *DB2 11 for z/OS What's New,* GC19-4068
- *Workload Scheduler for z/OS Customization and Tuning, SC32-1265*
- *Workload Scheduler for z/OS Managing the Workload, SC32-1263*
- *z/OS DFSORT Application Programming Guide,* SC23-6878
- *z/OS MVS JCL Reference,* SA23-1385
- *z/OS MVS JCL User's Guide,* SA23-1386

## Online resources

These websites are also relevant as further information sources:

- IBM Knowledge Center

  http://www-01.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/com.ibm.db2z11.doc/src/alltoc/db2z_11_prodhome.html
- Mainframe Performance Topics blog

  https://www.ibm.com/developerworks/community/blogs/MartinPacker/?lang=en

## Help from IBM

IBM Support and downloads

**ibm.com**/support

IBM Global Services

**ibm.com**/services

# Optimizing System z Batch Applications by Exploiting Parallelism

**Managing your batch window**

**Optimizing batch technology**

**Studying a hands-on case**

This IBM Redpaper publication shows you how to speed up batch jobs by splitting them into near-identical instances (sometimes referred to as clones). It is a practical guide, which is based on the authors' testing experiences with a batch job that is similar to those jobs that are found in customer applications. This guide documents the issues that the team encountered and how the issues were resolved. The final tuned implementation produced better results than the initial traditional implementation.

Because job splitting often requires application code changes, this guide includes a description of some aspects of application modernization you might consider if you must modify your application.

The authors mirror the intended audience for this paper because they are specialists in IBM DB2, IBM Tivoli Workload Scheduler for z/OS, and z/OS batch performance.

REDP-5068-00