CHAPTER 9

Determining a Suitable Test Mix

Finally, we are in a position to give attention to one of the core matters surrounding testing and tuning: determining a suitable stress-test business process or other "input data" test mix. After all, it is the mix of activities, transactions, and processes executed under your guidance that ultimately simulates your financials' month-end close or helps you understand the load borne by your SAP customer-facing systems during the holidays or other seasonal peaks. And it's the test mix that brings together master data, transactional data, customer-specific data, and other input necessary to fuel a business process from beginning to end. You've certainly heard the phrase "garbage in, garbage out." It applies without question here, because poor test data will never allow you to achieve your testing and follow-on tuning goals.

Beyond SAP application-level data, though, input data can also consist of the scripts, batch files, configuration files, and so on required of lower level testing tools, like those associated with testing the performance of your disk subsystem, network infrastructure, and so on. As you know by now, sound testing of your SAP Technology Stack encompasses much more than strictly business process testing.

The goal of this chapter is therefore to help walk you through the challenges surrounding data: how to select appropriate data, what to look for, and what to avoid. In this way, you'll be that much closer to conducting stress-test runs that not only "work" but also truly simulate the load planned for your production environment.

9.1. Overview—What's a Mix?

What exactly is a test mix? In true consulting fashion, the right answer is "it depends." The next couple of pages provide a high-level overview of a test mix, followed by more detailed discussions on this subject. For beginners, note that all test mixes must include a way to control timing, or the time it takes for a test run (or subtasks within a run) to actually execute. Sometimes this factor is controllable via the test mix itself or a test-tool configuration file, whereas other times it's left up to test-tool controller software or the team executing and monitoring the test runs.

Timing is not everything, though. Test mixes also typically allow the number of OS or other technology stack–based threads or processes to be controlled, another key input factor relevant to multiuser stress testing. In this way, the very load placed on a system may be controlled, making

it possible to vary workload rates or individual users or processes without actually changing the absolute number of users or processes.

Unfortunately, we must also keep in mind that the assorted tools and approaches we have examined thus far differ greatly in terms of their fundamental execution and goals, and therefore their specific data-input-mix requirements. On the flip side, test mixes tend to adhere to a set of general rules of thumb, too. But rather than trying to lay down a dictionary-style definition, let's instead look at test mixes by way of example, working our way up the technology stack, as follows:

- For network infrastructure testing, a test mix consists of factors like the number of data transfers or other network operations performed, the size of those transfers/operations, the use of configuration files that define the two end points necessary for testing (i.e., server-to-server testing), and even the protocols that may be tested (although for SAP testing, there's usually little reason to test anything other than TCP/IP-based RFC, CPIC, and ALE network activity).
- For disk subsystem testing, a test mix defines the number of reads and writes to be executed (either as a ratio or an absolute number), the types of operations to be executed (sequential versus direct/random, or inserts versus appends), data block sizes, and even the number of iterations (rather than leveraging timing criteria, a test mix might instead specify the number of I/O operations each thread, process, or test run will execute, such as 1,000). And the number of data files or partitions against which a test is executed is also often configurable, as is the size of each data file. For instance, I often execute tests against six different data files spread across six different disk partitions, each 5GB in size, thus simulating a small but realistic 30GB "database."
- For server testing, a test mix might include the number of operations that specifically stress a particular subsystem or component of the server (e.g., the processor complex, RAM subsystem, system bus). Server testing often is intertwined with network infrastructure and disk subsystem testing, too, and therefore may leverage the same types of data input as previously discussed.
- At a database level, a test mix must reflect operations understood and executable by the explicit DBMS being tested. Thus, SQL Server test mixes may differ from Oracle in terms of syntax, execution, and so forth. But, in all cases, a database-specific test will reflect a certain type of operation (read, write, join) executed against a certain database, which itself supports a host of database-specific tuning and configuration parameters.
- A single R/3 or mySAP component's test mix will often reflect discrete transactions (or multiple transactions sequentially executed to form a business process) that are executed against only the system being tested. I call these "simple" or "single-component" tests, because they do not require the need for external systems, and therefore CPIC-based communications, external program or event-driven factors, and interface issues stay conveniently out of scope. (Note that CPIC, or SAP's Common Programming Interface Communication protocol, allows for program-to-program communication.) The input necessary to drive these simple tests is transaction-specific, therefore, and rela-

tively easy to identify: all input data that must be keyed into the SAPGUI (anything from data ranges to quantities to unique transaction-specific values like PO numbers, and more) and any master, transactional, or other client-specific data must be known, available, and plentiful. User accounts configured with the appropriate authorizations and other security considerations also act as input. The right mix essentially becomes a matter of the proper quantity of high-quality input data versus the amount of time a test run needs to execute to prove viable for performance tuning.

- Complex R/3 or mySAP component test mixes, on the other hand, necessitate multiple components or third-party applications to support complex business process testing that spans more than a single system. Even so, the bulk of the input data may still be quite straightforward, possibly originating in a single core system. But like any complete business process, the *output* from one transaction typically becomes the input to a subsequent transaction. Cross-component business processes therefore require more detailed attention to input data than their simpler counterparts, because the originating system (along with other supporting data) must be identified as well.

Many of these examples are discussed in more detail later in this chapter. Suffice it to say, though, that a single TU or stress-test run can quickly grow complex from an input data perspective if several technology stack layers or systems are involved, especially in light of the many stack-specific test tools that might be used.

9.1.1. Change-Driven Testing

It's important to remember one of the fundamental reasons behind testing—to quantify the delta in performance that a particular change or configuration creates. I refer to this generically as change-driven testing, and believe that most if not all performance-oriented testing falls into this big bucket. But it's impossible to quantify and characterize performance without a load behind the change. In other words, things don't tend to "break" until they're exercised. The strength and robustness of a solution remains unproven, just like the maximum load a weight-trainer can lift, until it (or he or she) successfully supports or lifts it. The "it" is the workload, in essence the manipulation or processing of input data. Thus, it only follows that a good performance test depends on a good mix of data, data that has been identified and characterized as to their appropriateness in helping you meet your simulation and load-testing goals. Not surprisingly, stress tests executed without the benefit of adequate test mix analysis, or workload characterization, often fall short in achieving their goals.

9.1.2. Characterizing Workloads

Given that the workload a test run is to process is central to the success of stress testing, characterizing or describing that workload is paramount as well. Workloads vary as much as the real world varies, unfortunately, so there's no easy answer. Key workload considerations as they pertain to the SAP application layer include the following:

- The mix or ratio of online users to batch jobs, reports, and other similar processes, reflecting a particular business scenario.
- The mix of functional areas, such that the resulting mix reflects a sought-after condition (e.g., month-end closing for the entire business) or state (e.g., an average daily load).
- The pure number of online users, batch processes, report generators, and so on leveraged to create a representative load on an SAP system.
- The predictability of a workload, so that apples-to-apples comparisons may be made against subsequent test runs—in other words, avoiding true randomization when it comes to input data is important. True randomization needs to be replaced instead with pseudorandom algorithms that are at once "random" *and* repeatable.
- The quantity of data. Low quantities result in higher cache hit rates as more data may be stuffed into hardware- and software-based caches, therefore exercising the disk subsystem less than the workload truly would otherwise.
- The quality of data. Only unique combinations of customers, materials, plants, and storage locations "work" to create a sales order, for example. Other combinations result in error messages in the best of cases, and more often in failed or simply incompletely executed business processes.

The configuration of the system itself in relation to performance also represents "input" to some extent as well, though I prefer to keep this separate, under the guise of configuration or current-state documentation as discussed in the previous chapter.

But how do you determine what a representative workload or data mix looks like? In the past, I've spent most of my time speaking with various SAP team members to answer this question. Functional leads are your best bet, because they know the business as well as anyone else on the SAP technical team. But SAP power users representing each of the core functional areas are also valuable sources of workload information, as are management representatives of the various functional organizations found in most companies.

Outside of people resources/team members, you can also determine the "Top 40" online and batch processes through CCMS's ST03 or ST03N, both of which display detailed transaction data over defined periods of time. That is, you can drill down into the days preceding or following month-end to identify precisely the transactions that are weighing down the system most in terms of CPU, database, network, and other loads, along with relative response-time metrics. Particularly heavy hours, like 9 a.m. to 11 a.m., or 1 p.m. to 4 p.m., can be scrutinized closely as well. These "top transactions" can be sorted in any number of ways, too: by the total number executed, the peak hour executed, and even by greatest to least impact (highlighting the transactions that beat up the database or CPU or network the worst, for example). And, because CCMS since Basis release 4.6C allows you to globally view ST03 data across an entire SAP system, the once time-consuming job of reviewing individual application servers for your production BW system, for example, is no longer necessary.

Once you understand the particular transactions and business processes representative of a particular workload or regular event/condition, you must then consider how you will represent this workload in a stress test. To make the process of workload characterization as flexible and manageable as possible, I like to create "packages" of work. Within a package, the work is typically similar in nature—online transactions that focus on a particular business process or functional area, batch processes that execute against a particular set of data or for a similar period of time, and so on might make sense for you. Besides maintaining a certain functional consistency, I also chop up the packages into manageable user loads. For example, if I need to test 600 CRM users executing a standard suite of business activities, I'll not only create a set of business scripts to reflect those activities, but I'll also create perhaps six identical packages of 100 virtual users each, so as to easily control and manage the execution of a stress-test run. If the workload needs to be more granular and represent five core business activities, I might divide the packages instead into these five areas (where each package then reflects unique business process scripts), and work with the business or drill down into the CCMS to determine how many users need to be "behind" or associated with each package. Even then, if one of these granular and functionally focused packages represents many users, I would still be inclined to chop it up further as just mentioned and as shown in Figure 9–1. More details regarding the creation and divvying up of test packages are discussed later in this chapter.

9.1.3. Don't Forget to Baseline!

Beyond baselining and documenting the configuration to be stress tested, each test mix also needs to be characterized by way of a baseline. The baseline serves as documentation relevant to a par-

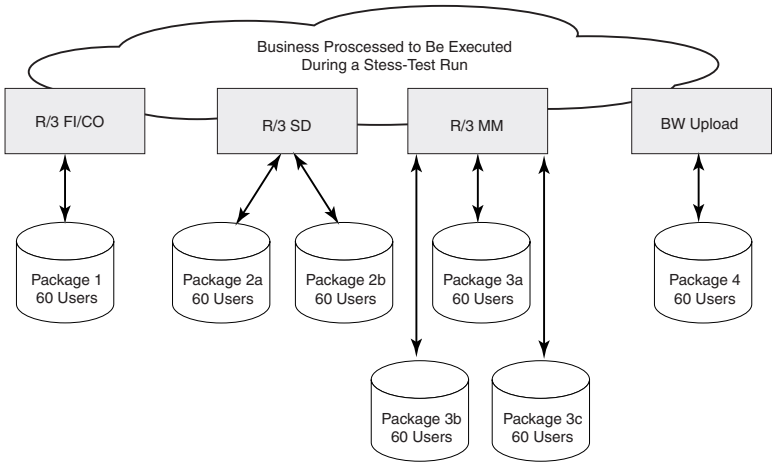

Figure 9–1 Once your workload is characterized and sorted by tasks, functional areas, or business processes, it makes sense to further divide the workload into smaller and more manageable packages, each reflecting a representative user mix.

ticular state—the configuration of the system in relation to the workload that it can support—so that future changes to either the technology stack or workload are not only easily identified but also specifically quantified in terms of performance. In my experience, these initial performance baselines tend to multiply rather quickly. For instance, I'll often begin initial load testing by working through a number of specific configuration alternatives, testing the same test mix against each different configuration in the name of pretuning or to help start off a stress-test project on the best foot possible. Each alternative rates a documented baseline, but only the most promising end-results act as a launch pad for further testing and pretuning iterations.

Once a particular configuration seems to work best, I then move into the next phase of baselining, called *workload baselining*. This phase is very much test mix–centric, as opposed to the first phase's configuration-centric approach. The idea during these workload characterization and baselining processes is to maintain a static system configuration (i.e., refrain from tuning SAP profiles, disk subsystems, etc.), so as to focus instead on monitoring the performance deltas achieved through executing different workloads. All of this is performed by way of single-unit testing, of course, to save time and energy. The goal is quite simple, too: to prove that a particular workload indeed seems to represent the load described by the business, technical teams, or CCMS data. The work of scripting a true multiuser load followed by real-world performance tuning comes later, then, after you've solidified your workloads and executed various stress-test scenarios as depicted in the next few chapters.

Baseline testing is useful from a number of different perspectives. For instance, it's useful even when it comes to testing a non-Production platform, like your Development or Test/QA systems deployed prior to building, configuring, and initially optimizing the Production platform. That is, you can still gain a fair bit of performance-related knowledge testing against a non-Production system, both in terms of single-unit and multiuser stress testing. And this knowledge can pay off simply by helping you create a faster development experience for your expensive ABAP and Java developers or by creating a functional testing environment for a company's unique end-user base. This is especially true prior to Go-Live, when the production environment is still being built out, but the need for initial single-unit or "contained" load testing is growing ever more critical. The same goes for any change-driven testing as well, however, such as that associated with pending functional upgrades, technical refreshes, and so on.

9.2. Real-World Low-Level Technology Stack Test Input and Mixes

Although we have looked at a number of input data and test mixes at the highest application levels, the insight only made possible from real-world low-level examples should help lay to rest any lingering questions as to the value of your input data in executing a stress test. The next few sections detail specific stress tests from an input data perspective, identifying goals and how specific stress-test tools help achieve those goals along the way.

9.2.1. Input Required for Testing Server Hardware

Many testing tools designed to validate the performance of your server platforms require little if any formal input. CheckIt requires nothing other than installation on the local machine, for ex-

ample. But if your goal is to compare one Windows-based SAP server platform to another in terms of CPU performance, CheckIt allows you to "save" the testing results yielded from one platform's test as comparison or baseline data against which other platforms may be measured and compared. Thus, the output of one test acts as input, in a manner of speaking, for subsequent tests.

Other tools offer a bit more granular control, however, and not surprisingly, support additional data input metrics. If your goal is to measure how well your memory subsystem handles operations of different sizes, or differing random or sequential accesses, Nbench is an easy solution. It features the ability to customize some of your input, specifically the following:

- The size of various operations. Thus, you can test operations that reflect what your SAP server does (or will do) in production, rather than rely on arbitrary hard-coded values assumed by other tools. You might test your DB server for large values, for example, while you focus on smaller values for applications and Web servers.
- Execution thread count. Again, this gives you flexibility to emulate the expected level of multiprocessing that occurs in (or is expected of) your unique SAP environment, specifically regarding the disk subsystem.
- Values associated with integer, floating point, and memory operations.

Other input values unlikely to require customization include the access width of the memory bus (tested at a number of different levels), timing/control information, and the performance of both random and sequential operations. Other mainstay general hardware test tools, like IOzone, even go so far as to set processor cache size and line size to particular default values, though both of these settings and much more may be changed through command-line or switch settings.

9.2.2. Disk Subsystem Test Mixes

Disk subsystem test mixes, like those associated with using SQLIO, Iometer, IOzone, and others have quite a few things in common. First, the period of time a test will be executed is controllable through switch settings or manually through the standard "control-c abort" sequence. Second, the size and location of one or more data files (representing one or more "database" files) is configurable as well, as is the mix of reads to writes and ratio of sequential operations to random/direct operations. These tools also allow you to control the number of processes or threads utilized by the OS to execute the test, in effect allowing you to control the disk queue lengths that must eventually be processed by the OS and its underlying disk controllers and drives. Finally, this and the settings for many other switches can be saved in a single "input" configuration file, useful in ensuring consistency between iterative tests executed against different hardware or software configurations.

Of course, if we step back and analyze the need for input at all levels, the fact that a particular disk stress-test utility may only support a specific OS version, patch level, or similar operating environment factor reflects core input data as well. And if the output is only provided in a particular format, the installation of special readers or a specific version of Microsoft Word or Ex-

cel may be warranted, too. Along the same lines, if a particular subset of output data is also desired—like the CPU and system utilization performance metrics that can be captured and shared via SQLIO—the appropriate switch needs to be manually set (in this case, the "-Up" switch).

9.2.3. OS Testing and Tuning

Performance testing the configuration options available to a particular OS often boils down to making specific utility– or command-line–driven changes and executing before and after test scenarios that indeed help quantify any change in performance. For example, I've done extensive testing in the past on pagefile sizing for SAP R/3 Windows-based database and application servers, ranging from release 3.0F through 6.20. SAP's recommended guidelines changed quite a bit in the days just before Basis release 4.0 was made available, and continued to change somewhat over the last two years as well. My goal was to determine which configurations made the most sense from both financial and performance perspectives. To this end, I leveraged different hardware configurations (e.g., disk controllers, use of RAID 1 versus RAID 5, use of multiple disk spindles versus a pair of drives) as well as different pagefile sizes, distribution models, and so forth. At the lowest levels, I used the Compaq System Stress-Test tool (once known as the Thrasher Test Utility) to force paging operations and therefore establish a baseline reflecting the relationship of I/Os per second for a particular memory range to the level of Windows paging that resulted. In a similar way, I also tested the impact that the "maximize throughput for network settings" Windows setting had on the memory subsystem and OS in general. In both cases, the tools I used for application-layer testing were nothing more complicated than custom-developed AT1 scripts. To create and drive a repeatable and consistent application-layer load, I simulated 100 users with minimal think times executing a suite of simple though typical R/3 transactions (e.g., MM03, VA03, FD32, and others that only required a single SAPGUI input screen). I could then compare the low-level thrasher results with the high-level SAP application-layer results, and extrapolate how different workloads would impact memory management in general and paging in particular.

9.2.4. Test Mixes and Database Tuning

At the simplest levels, a database stress test begins with read-only queries that preferably execute against a copy of your actual production database (though a copy of a development or test client or even a small sample database may suffice, depending on your goals). Unless you want to bring in the entire application layer of your SAP system, you should consider a number of testing alternatives. For instance, tools like Microsoft's SQL Profiler allow SQL 7 and SQL 2000 database transactions to be "captured" at a low level, and then replayed against a point-in-time database snapshot at a later date, all without the need for SAP application servers, an SAP Central Instance, or anything else that is SAP-specific. This type of approach is perfect for stress tests where an organization assumes (and rightly so most of the time) that potential application-layer performance issues can generally be solved either by adding more application servers or by beefing up the existing servers. That is, performance at the application issue is moot and can therefore be pushed out of scope simply because SAP supports a robust horizontal scalability model in this regard.

So the test mix for a pure DB-based stress-test scenario simply involves the use of record and playback tools—database-specific tools capable of capturing the SQL statements executed during a certain timeframe by a representative group of end users. You might choose to capture the busiest day of the season, for example, or maybe the 4 hours during which the heaviest batch job load is being processed. The queries, table scans, joins, and so on captured during this time period become your repeatable set of input data, to be played back for the DB server—you need not concern yourself with your SAP instances.

These types of tools are necessarily database-specific, of course. That is, Informix administration tools simply cannot support Microsoft SQL Server, nor do SQL Server–based tools support Oracle. But the value of these record/playback tools is unquestionable—they're generally quite easy to learn and easy to use. And, because they tend to support the ability to play back the captured transactions in the exact timeframe they were recorded, or compress or stretch out the timeframe as you see fit, their value in terms of capacity planning and what-if analysis is great.

Case in point, as I mentioned before I used SQL Profiler to record the real-world transactions of one of my large R/3 customers that was preparing for an acquisition that would double the number of its online and concurrent users. The company was comfortable with the scalability of the application server tier of their current solution, but needed to better understand the impact on the DB server. To be sure that this SQL Profiler approach to testing would be suitable, I first went through a sizing exercise, analyzing the current mix of users (via transactions ST07 and AL08) in terms of the total number of users as well as the functional areas in which they were heaviest. I then analyzed similar data provided by the organization to be merged into the first, and found that the mix was close enough to warrant no further analysis—within plus or minus 10%, each organization supported about the same number of SD, MM, PM, FI, and CO users. I used these real-customer data to tweak my SAP sizing tools, to better reflect the weight of their as-configured functional areas (rather than relying on default weights provided by the tool), using data gleaned from Basis transactions ST02, ST03, ST04, and ST06. This allowed me to size a new DB server, which I then procured from our seed unit pool (a pool of gear used for the express purpose of customer demonstrations and proof-of-concept engagements). After loading the OS and database, configuring the server, and then working through a restore of the customer's 400GB SAP R/3 database, I was finally in the position to do some testing.

It was at this point that the value of SQL Profiler really sank in—because the number of users would double in the new environment but the mix of users would stay pretty much the same, I simply sped up the playback of my previously recorded transactions 2× and then sat back and observed how well my newly architected and deployed demonstration DB server handled the load. In this way, I was not only able to nicely simulate the customer's actual expected load, but I was able to validate our user-based sizing as well. In the end, after making some calculated processor and RAM upgrades to the currently deployed application servers (to account for the additional logged-in users), and installing a new DB server identical to the one I tested back in the lab, the customer's acquisition went off without a hitch. This database-centric approach to testing and tuning should appeal to any SAP test team focused on saving time and money when the other

tiers of a technology stack can be ruled out and it's determined that full-blown end-to-end stress testing is simply not required.

Beyond these basic database-level test mixes, where only the raw SQL code is executed on a DB server, lies the ability to execute application-driven SAP transactions. Such transactions must be commenced on an SAP front-end and executed by an SAP Application Server, of course. But this incremental complexity gives you the advantage of testing the impact that your test mix places on your end-to-end solution. And, by varying this test mix, which might range from light-impact financial and material-based transactions to multicomponent and truly solution-intensive monster batch transactions, you can exercise various solution stack layers with near pinpoint accuracy. Detailed test mixes and the challenges a team faces in identifying and using them are covered in detail near the end of this chapter.

9.3. Testing and Tuning for Daily System Loads

Load testing the average daily expected workload is perhaps the most fundamentally overlooked type of performance testing I'm aware of. The companies that embrace performance testing and tuning tend to focus instead on high-water peak stress testing or saturation-level smoke testing, or spend their budget dollars tweaking and tuning the disk subsystem. Certainly, these other areas are critical, but the first question that comes to *my* mind is "How often do you see the loads associated with seasonal peaks?" followed by "How truly important is it to know precisely at what transaction load your system breaks?" when it has never even been tuned for the mundane daily workload processed 99% of the time.

Tuning for your anticipated daily transaction load should represent the starting point for SAP application-layer stress testing, and used, as depicted in Figure 9-2, as a launch pad for in-

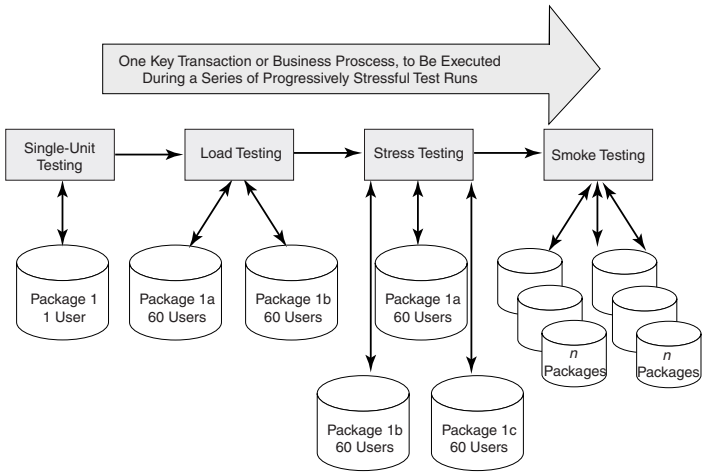

Figure 9–2 Use single-unit load testing as a springboard for daily load testing, peak stress testing, or high-water smoke testing.

cremental load, stress, and smoke testing. Specifically, you should have a series of TUs or test packages that, when combined, can emulate the typical online and batch workload seen Monday through Friday, 9 a.m. through 5 p.m. In this way, subtle changes to your workload or configuration alike can be quickly simulated, providing the kind of rapid feedback necessary for supporting sound change control processes. And the tuning made possible will benefit you 99% of the time, huge by any measure, because your system is optimized for a relatively steady workload. Finally, understanding and capturing your daily system load pays big dividends when it comes time to perform a major upgrade, add or remove a significant component of the workload (acquisitions and divestitures), or simply build on the load in support of testing and tuning for various business peaks, covered next.

9.4. Testing and Tuning for Business Peaks

Though the tuning made possible by daily expected workload testing will benefit you 99% of the time, you must still work through an ROI exercise designed to determine whether it's financially advisable to stress test the remaining 1% of scenarios. To an outsider, 1% might seem trivial, but if the 1% represents any of the following scenarios, the financial impact associated with planning for, setting up, configuring, executing, monitoring, analyzing, and tuning might easily be justified:

- If you bring in a significant portion of your revenue via seasonal peaks
- If you must process a time-sensitive workload in a particular execution window (e.g., 2 hours start to finish), such as payroll checks or tax-return data, the failure of which would produce financial or other significant losses
- If specific SLAs are in place that penalize you significantly for "missing" key response-time or throughput metrics
- If your business peaks represent an opportunity to compress financial cycles or make it possible to gain an edge on the competition in terms of insight into product sales, order fulfillment, inventory turns, and so forth.
- If customer satisfaction is impacted beyond what is otherwise grudgingly acceptable

Business peaks vary nearly as much as the companies that endure them. As I mentioned before, almost every organization must suffer through some kind of seasonal peak workload—testing and tuning for this peak can mitigate much of the suffering. Similarly, working through issues surrounding weekly, month-end, and quarter-end financial closings, warehouse inventories, payroll runs, HR benefit update windows, and so on will not only help you avoid self-imposed or externally driven SLA penalties but also provide a competitive long-term advantage. And like the value of knowing your daily load, understanding your business peaks positions you that much better when it comes time to perform a major functional upgrade or technology refresh.

Data input required for your business peaks should not be that difficult to identify, because it's almost always associated with a particular business activity or functional area, as we have already seen. But it's all about *quantities!* The key is to ensure you have *enough* data to do the job

of stress testing, along with access to the proper number and mix of virtual or physical users that truly represent the concurrent online or batch load you seek.

9.5. Identifying Key Transactions and Business Processes

As I said before, there are many ways to identify the key transactions and business processes representative of a particular condition, event, or state. The opinions and experience of the end-user community that actually conducts much of this work is an obvious source of knowledge, albeit an imperfect and not necessarily holistic source. Regardless, I still recommend that you speak with your business representatives and functional experts. Indeed, for systems that have not gone "live" yet, there's really no alternative. But for precise and historically accurate information that reflects the load borne by your production systems day in and day out, week after week, month after month, I strongly encourage you to leverage the abundance of transactional and performance data sitting in CCMS. In this way, you can have insight into the whole picture—the most popular online transactions, the heaviest batch processing jobs and execution windows, visibility into the mundane repetitive tasks collectively responsible for much of the load, and so on. Month-end closes, seasonal peaks, and other high-water loads can be clearly understood in this way, and beyond this, how all of the various functions intertwine and come together to create the lifeblood of an SAP system—its workload—becomes clear as well.

Prior to following the steps outlined in the sections that follow, you need to determine the scope—typically by looking at an entire collection of application servers servicing an SAP system—and then the timeframe you wish to analyze. That is, SAP CCMS will allow you to look at the last "X" minutes or, probably more applicable for our purposes, a particular time period. I often start with analyzing the load of what I'm told is a "typical week" by plugging in start and end dates that reflect 7 full days (usually Sunday at 3 a.m. until the next Sunday at 2:59 a.m., though the timeframe you select may be different). I like somewhere around 3 a.m. because it's often the quietest time of the evening—backups are usually completed, the system is back up and available, but scheduled early-morning batch processes have not yet commenced, and few online users tend to be on the system. I try to avoid capturing only a partial business process or workload—as much as possible, I want to capture the entire week's work the moment it begins, without cutting anything off in the beginning or the end.

From this starting point, I then work to identify peak days within the week, and even peak hours within particular days. The term *peak* is subjective, of course, but most often involves first uncovering and sorting the quantity of transactions executed. Next, I'll change tack and look at all the transactions sorted by database load, CPU load, and so on. In this way, I can begin to understand the load placed on the various hardware components that underpin SAP. After this detailed analysis, I'll often take a look at an entire month's worth of data as well, just to be sure I didn't miss a particular processing peak not easily seen otherwise. And, in some cases, I might even drill down into a particular application or batch server, especially if capacity planning is one of the goals of the stress testing. In the sections that follow, we will walk through the exact steps necessary to uncover the specifics that come together to create your workload. To help you apply

this process to all SAP systems, I will only draw on CCMS, as opposed to SAP Solution Manager or third-party systems-management applications and other tool sets which you may or may not have at your disposal.

9.5.1. Online User Transactions

To determine the mix of your peak online user transactions and how they interplay to create a load on your SAP system's hardware components, log on with a user ID capable of executing core basis transactions to the system being tested, and perform the following steps (if you prefer the newer ST03N, keep in mind that a certain amount of modification to the listed steps will be required. The changes are quite intuitive, fortunately, and for experienced SAP administrators or Basis consultants will present few problems, if any):

1. Execute transaction /nST03 (the Workload Analysis screen is displayed).
2. Click the "Performance Database" button.
3. Select timeframe.
4. Select the "dialog" button—in this way, only online transactions are analyze.
5. Select "detailed analysis."
6. Select a column to sort by, such as CPU, and double-click it. Transactions will now be sorted by those that consume the most CPU time. Record the top 40 transactions.

9.5.2. Batch Processing and Reporting

Follow a process similar to that outlined previously, though changed to reflect batch processes:

1. Execute transaction /nST03 (the Workload Analysis screen is displayed).
2. Click the "Performance Database" button.
3. Select timeframe.
4. Select the "background" button—in this way, only batch processes are analyzed.
5. Select "detailed analysis."
6. Select a column to sort by, such as database response time, and double-click it. Transactions will now be sorted by those that are most disk-intensive. Record the top 40 transactions.

9.5.3. Most Popular Transactions and Other Workloads

Using ST03, the process to identify the most popular transactions (i.e., those that aren't necessarily the hardest hitting but represent the bulk of activities performed on the system) is as follows:

1. Execute transaction /nST03 (the Workload Analysis screen is displayed).
2. Click the "Performance Database" button.

3. Select timeframe.
4. Select the "total" button—in this way, all transactions are analyzed.
5. Select "detailed analysis."
6. Select a column to sort by, such as CPU, and double-click it. Transactions will now be sorted by those that consume the most CPU time. Record the top 40 transactions.

9.5.4. Mixed-Bag Testing

With the step-by-step processes outlined previously, you should now have an understanding of the key online, batch, and noise transactions hosted by your system. At this point, you need to bring this knowledge together to create a "mixed-bag" workload—in effect, your workload to be emulated through scripting. Your workload mix will vary based on the goals of your test. For example, if you wish to test the benefits of a new server platform, you'll probably want to focus on the transactions that drive the heaviest CPU load. Similarly, if an updated disk configuration or brand-new virtual SAN is potentially in your future, you'll probably wish to create a workload that beats up the disk subsystem in the most representative manner.

Refrain from only focusing on a single type of transaction, though, if time and budget allow. Variety should be your mantra. For example, when it comes to disk subsystem testing, I suggest that you combine a number of hard-hitting online transactions and reports along with key batch processes, rather than simply going with easily scripted batch loads (easy because once you do one type, others tend to follow similar patterns or allow for cookie-cutter scripting approaches). In this way, different disk access patterns will be represented, which in the end will also best represent the real world. And I suggest tossing in a few noise scripts as well simply to keep a certain level of constant activity in the background, again representing the real world of most SAP enterprise solutions. At the end of the day, a mix of perhaps 50% batch processes to 25% online transactions and reports and 25% scripted noise activities will serve you well in stress testing and tuning your SAP disk subsystem. And the same argument can be made for testing other subsystems or technology stack layers, too—variety is desirable as long as your budget can absorb the incremental time necessary to perform the requisite business process scripting.

9.6. Real-World Access Method Limitations

Although I've focused on test mixes from an application or low-level technology stack perspective thus far, another area that needs to be considered relates to the front-end SAP client. SAP offers a number of access methods or alternatives: the classic SAPGUI, which is available for a number of "desktop" platforms is the most prevalent, of course, followed by the HTML-based WebGUI and the newer JavaGUI. Generally speaking, though, the following challenges relevant to input mixes and access methods apply across the board:

- Not all test tools support all front-end client interfaces. Thus, test input quickly becomes a moot point if your goal is to perform virtual multiuser SAPGUI-driven testing but the available tool does not support the SAPGUI. The same holds true if you wish to

leverage the WebGUI, but your preferred tool is incapable of driving Web-based content.

- Similar to the previous point, the specific development tool and mySAP component-specific SAPGUI snap-in may or may not be supported by a particular testing tool.
- Not all business transactions are supported by all user interfaces, especially those outside of the classic SAPGUI. This is not nearly the problem today as it has been in the past, where a small percentage of a business's key transactions simply could not be executed via the WebGUI (because of the absence of HTML-based transaction equivalents of the standard RFC-based transactional data and screens).
- Not all access methods are hosted directly by a client PC or laptop. For instance, a number of my larger SAP accounts have standardized on Citrix MetaFrame solutions for hosting and controlling the SAPGUI. Although the benefits are huge in terms of standardization, lower desktop software upgrade and maintenance costs (it costs an average of $40 for IT just to "touch" a user's PC—that is, to upgrade the SAPGUI), and extending a user community's PC lifecycle, the costs to replicate the necessary Citrix server infrastructure in a test environment can be prohibitive.
- Not all functionality is available on particular releases of the SAPGUI. For example, only the latest SAPGUI releases support some of the newest business and Basis transactions. And, even previously, the use of OCX controls and other GUI features were only supported on the SAPGUI releases provided after the "EnjoySAP" initiative.

The lesson here should be clear—take the time up front to determine how well your goals and success criteria affect the technology stack that is being tested. And then perform some basic front-end testing to ensure that what you envision is indeed possible with the tools, time, and expertise you have available.

9.6.1. Other Front-End Components and Interfaces

Front-end user interfaces outside of those previously discussed can come into play as well. The SAPGUI snap-ins mentioned earlier may or may not be supported by your favorite virtual testing tool, for instance. In fact, the interface you need to use in support of a specific TU may be completely foreign to SAP. Compuware's TestPartner is a good example of a tool that can prove helpful in these cases—not only is it a great WebGUI test tool but it also allows you to test a variety of user interfaces outside of those dependent on Internet Explorer or similar Web interfaces.

In these cases it is important to look beyond what is perceived as immediate front-end needs, and instead take a quick look at alternatives. Perhaps your testing can be driven without the need for a GUI—SAP eCATT can "talk" directly with SAP behind the scenes; or maybe part of the access-enabling technology that sits in front of your SAP system represents an unnecessary stumbling block, something in the way of your testing goals. If you need to test the online user load your R/3 system can handle, but EP represents the primary access vehicle to R/3, perhaps it's not absolutely necessary to keep EP in the testing loop, so to speak. In other words, to make

things simpler you might consider forgoing the inclusion of EP into the solution stack to be tested back in the lab, and use the simpler direct method of access allowed by virtual SAPGUI connections. In this case, you won't be in a position to test the ability that your particular system has to support single sign-on (SSO), nor will you be able to validate any high-availability or other performance or scalability metrics. The same limitations hold true for SAP's older portal product, Workplace, as well as other legacy tools and utilities that help make SAP system access possible. Bottom line, though, if your goals don't require it, you can make your test infrastructure and test runs that much simpler by keeping only the infrastructure and components necessary to meet your test's goals or success criteria in scope.

9.7. Best Practices for Assembling Test Packages

With regard to breaking down and assembling good test packages, I've already mentioned a number of key approaches I often use. Four of these are worth more discussion, however, and are covered next.

9.7.1. User-Based Test Packages

A simple method of controlling the amount of "noise" or any other functionally derived activities you introduce in a stress-test run is to directly control the number of users tossed into the mix. With this approach, you're not concerned about the load placed on the system per se, you're more concerned with the number of users hosted, and whether each user is executing consistent and repeatable work. I've mentioned elsewhere that I like to create packages of 100 users. One hundred is a nice number simply because it's a round number; its size is significant enough to allow me to build up a test run into a "large stress test" very quickly and in a highly controlled manner.

Load becomes an issue only if a group of 60 or 100 users (or whatever number you choose) overtaxes a system, invalidating the stress-test run. An obvious example might involve executing custom Z reports—if you toss 100 of these into a system configured for only half as many batch processes, you'll simply create a queue of work that monopolizes all batch work processes and completely destroys most DB servers. A better number in this case might be 10 instead. Experiment to find the best number for you.

9.7.2. Functionally Focused Test Packages

Although controlling the pure number of users participating in a stress test makes sense, taking this to the next level and controlling a group of 100 SAP R/3 SD users, for instance, or 10 BW custom InfoCube reporting users, makes even more sense. Of course, you'll need to be careful to ensure that the mix of users (e.g., or batch processes, or reports) adheres to the mix you need to emulate in support of your test's specific success criteria. And your test tool needs to support both the high-water number of users you wish to simulate as well as the ability to create and control multiple packages.

9.7.3. Another Approach—End-to-End Business Processes

Building on the previous approach, this next approach is both intuitive and in many cases simply necessary. That is, because a business process by definition feeds off one transaction (the previous transaction's output data, actually), and then goes through a processing phase only to hand off newly created or processed data to the next transaction in line, the idea of bundling these transactions into a single package seems logical. Beyond this, though, it saves time and effort in scripting, too, because a common set of fewer variables can be leveraged. And a straightforward input-output approach to scripting lends itself to making even cross-component business processes more easily controlled than is otherwise possible. Finally, the granular control made possible through this method makes it easy to quickly ramp up the user count of a stress-test run while simultaneously ramping up the number of complete business processes to be executed.

9.7.4. Tips and Tricks—Making Noise with Noise Scripts

One of my favorite approaches to SAP scripting and stress testing involves the creation and deployment of *noise scripts*. As I said earlier, noise scripts capture and help represent the background processing or "noise" common in all SAP production systems. I typically create a variety of noise packages, some focused on general functional areas (e.g., MM or FI, where many lightweight transactions are common), whereas others might be focused on SAP Basis activities (to represent the load that monitoring places on a system), specific batch or report jobs, and so on. The key is to create a consistent baseline of user or batch-driven noise behind the scenes, and then quantify the per-package load to establish a tier of potential baselines, as depicted in Figure 9–3. Does this ancillary load represent 10% of the typical production workload? Or 20%? What is the impact of the load on your test hardware (that is, the HW hit)? Consider what many of my colleagues and I deem to be best practices as follows:

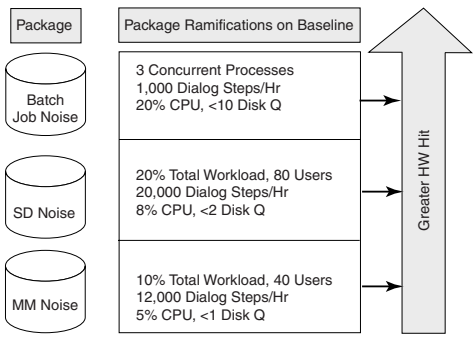

Figure 9–3 Noise scripts are useful in providing the fundamental underlying nonprimary transaction load underneath all productive SAP systems.

- Baseline just your noise scripts, to ensure they do the job you envisioned for them. Baseline not only SAP application-layer performance, but lower levels as well. Eventually, I recommend that you settle on any number of online users, batch processes, and so on that create an easy-to-measure load on the system, like 10% CPU utilization or disk queue lengths of three per disk partition or drive letter.
- Keep the target baseline utilization numbers small, so that it's easy to add incremental measurable load to a stress-test run—simply throw another package into the mix, for example, to add another 10% load on the CPU or perhaps another 40 users or three concurrent processes (whatever measurement you judge most valuable).
- Ensure that your noise scripts are pseudorandom. As I mentioned earlier, they need to be repetitive enough that they maintain a consistent load on the CPU, while random enough to encourage physical disk accesses. In other words, you don't want to create a noise script, or any script for that matter, that executes at different speeds every time it runs, or processes significantly different data between test runs. Make it repeatable!
- Ensure that you track the number of iterations executed, along with the specific number of discrete noise transactions executed within your noise script or scripts. This is useful after-the-fact, when you're seeking to understand and analyze a stress-test run—I suggest leveraging a counter of sorts within the body of your scripts (e.g., and publishing the counter's value to your output file), or simply dumping the script's output into your output file, to be counted in more of a manual manner after the run.
- Finally, if sound test management dictates that you should group your noise scripts together, you'll logically want to go to the trouble of creating one or more noise packages that either complement the core load being tested (you may create a noise script that effectively mirrors many of the transactions that represent core activities) or act as a "gap filler" and instead "round out" a test load (e.g., adding batch noise to a primarily online-user–based stress test).

One of the simplest methods of generating noise within a test run is to execute every core T-code twice—not the entire transaction, just the T-code associated with the first transaction in a business process. This kind of incremental and predictable load on the CPU is ideal when it comes time to measure overall performance, because the transaction is always executed from cache the second time it's executed. In this way, it not only does not ever disturb the system's buffer contents but it is easily scripted or added at the last minute in an iterative fashion if you need to bump up the CPU hit on a particular stress-test run.

9.8. SAP Component and Other Cross-Application Test Mix Challenges

For each particular SAP application, component, or solution, there tends to be a special set of circumstances or limiting factors that complicates creating a repeatable, consistent, or adequate test mix. The following list identifies key problem areas I've run into in the past, and what I did to work around these issues:

- R/3 business processes range from one-offs to multicomponent, highly complex sets of transactions. But, for most companies, R/3 activities tend to focus on discrete transactions run repeatedly by many users throughout the day, alongside a subset of core batch processes. I suggest focusing on a few key business processes, like order-to-cash, to take advantage of the input/output nature of the underlying transactions. In this way, obtaining input data is a simple matter of leveraging the previous transaction's output. And the first transaction—a sales order—can originate in any number of other systems and be easily completed if core customer, material, and plant data are available and abundant.
- APO includes both online users and batch-oriented business processes. The real load borne by the system is represented by the latter, though, especially with regard to demand or production planning runs, or Available to Promise (ATP) processing. Online user activity is negligible in comparison. From an input data perspective, the number of key figures that reside in your liveCache server, the number of characteristic combinations (properties) that describe an object, the number of periods (measured in weeks) against which these processes will run, and of course the number of sales, purchase, planned, and other orders transferred from R/3 to APO are all important.
- BW and SEM activities range tremendously, so it's no surprise that input data ranges are equally wide. In the past, I've started a BW stress test by kicking off an R/3 data extraction process, or simply by pulling data from the ODS to create and populate a cube. These are certainly important functions, but may not represent your most important goals. Instead, you may wish to run queries against one or more standard or custom InfoCubes. The key will therefore be the cubes themselves—if it matters little how the data move through your SAP system landscape, make it easy on yourself and start with a fully populated set of cubes.
- CRM data hail from a number of systems within the CRM system landscape, including the TRex Server, Multi-Channel Interface Server, InQMy Application Server, and potentially a Workgroup Server and Communication Station, not to mention the CRM server itself. As such, it can be one of the more complex SAP solutions to stress test. As a starting point, I suggest that you concentrate on the core user type your system hosts (probably a Mobile Sales or CIC user, though others exist), and determine the input necessary to support the top five or so key user transactions. For your Mobile Sales users, you might focus on managing opportunities or activities, creating customer orders, performing service-related transactions, or managing customer, product, or project-specific data, for instance. If you host CIC users, you might instead go with the transactions and activities outlined in the SAP CRM benchmark kit, for example, and focus on transactions relevant to managing incoming and outgoing CIC calls
- EBP (the core component of SRM) is quite complex, especially from an input perspective—data originating from Requisite BugsEye (an online catalog), R/3, BW, and EP complicate executing core EBP business processes. But that's not all! There are other

SRM components that need to be considered as well, like the SAP Bidding Engine and SAP Supplier Self Services. Given all of this, a seemingly straightforward shopping-cart–driven procurement business process will have many touch points, will pull data from many sources, and will generally represent a whole lot of scripting and coordination effort.

- EP has evolved significantly over the last few years and today represents one of the fastest growing products in SAP's line-up. It's also potentially one of the most important products a company will implement, in that all users could conceivably (and, it is hoped, will) leverage its single sign on capabilities to gain access to all other SAP systems and many third-party resources, creating a potential high-availability and performance nightmare should the EP technology stack lack scalability. Thus, stress testing EP will only grow in importance as it continues to be deployed across the globe. Fortunately, stress testing can be conducted quite easily. Sure, many systems represent potential integration points, but an effective load test could very well consist of accessing only local resources.
- PLM supports users responsible for managing product, asset, and process information at any point in the product lifecycle, from selection and purchasing through production ramp-up, installation, operation, engineering changes, maintenance/repair, retirement, and more. From a data input perspective, then, you need to understand the precise functionality being implemented and what then needs to be tested—everything from Lifecycle Data Management and Asset Lifecycle Management to core functionality like Program and Project Management, Quality Management, and Environment, Health, and Safety (or EHS) can come into play. Plus, because PLM is implemented as an enterprise portal solution and is tied closely with CRM and likely APO, the business processes that can result may be complex indeed. I therefore suggest going after either the biggest couple of functional areas to be implemented (in terms of transaction counts or user counts), or instead the most critical functional areas, and script core transactions rather than full-blown business processes.

The world of SAP has grown considerably over the last year, though—well beyond the components and solutions just highlighted. The SAP XI, xApps, and other products represent additional systems that may be tested individually or as part of a larger SAP solution. For example, simulating the number and size of the messages that the XI Integration Server processes represents an excellent method of testing XI without all of the integration points necessarily in the picture.

XI offers some exciting capabilities when it comes to viewing collaborative cross-application SAP business processes. For example, business processes that span your R/3, APO, and CRM systems can be tied together via synchronous and asynchronous messaging defined by and maintained within XI. You may then leverage XI to enable true cross-application business process stress testing, regardless of whether heterogeneous system landscapes have been deployed—test cases would initiate from one of the core SAP systems tied together in this manner, of course. And test execution would be seamless, because XI would handle moving the output

from one transaction into the proper component, where it becomes input for the next transaction in a business process. Beyond this awesome timesaving by-product of XI integration, XI also allows you to visually depict and manage SAP components as well as other enterprise applications from Baan, Broadvision, JDE World Software, Oracle, PeopleSoft, Siebel, and more. In this way, the inclusion of SAP and third-party enterprise applications may all be tied together into a cohesive virtual system. More to the point, these complex systems may be managed as a single entity rather than as a bunch of individual systems, which further simplifies testing.

Mainstays like SAP's ITS also continue to thrive. From a data input perspective, I suggest you focus on common transactions supported by back-end systems like R/3. Because all (or nearly all, in the case of older releases of R/3) SAP screen content is maintained in HTML on the ITS server, testing ITS amounts to testing the speed and throughput of Web connections. Fortunately, tools abound that support this input type, as we reviewed in Chapter 6, and scripts of this nature are fairly easy to write and maintain. And the best news of all is that you'll find that your ITS users tend to execute the same transactions as their SAPGUI-enabled colleagues (though exceptions exist—I've got a customer that has pushed the bulk of its HR and ESS workload on ITS, for instance, whereas other modules and functional areas are accessed via the traditional SAPGUI). So once you've established the core transactions executed by a particular SAP product or component, you're well on your way to using this information in support of ITS testing, too.

9.9 Tools and Approaches

Although there are few formal tools that help you analyze your input requirements, the approaches discussed throughout this chapter can save you time. I suggest that you document step-by-step processes (e.g., those pertaining to ST03 herein) specific to your TU, in effect documenting how you arrived at your determination that a particular set of input was both appropriate and adequate. Take a screen shot as well, and file this in your documentation repository. It may again prove useful as you iteratively execute various new tests, tune your system, and then return to testing.

Finally, take care not to assume too much when you find yourself implementing new SAP components or technology stacks because much of the documentation floating around is not necessarily applicable. For instance, with the advent of commodity 64-bit computing from Intel and AMD, there is a need for basic pagefile and OS memory/caching testing, along with database tuning as I described earlier in the chapter. This is the case because the rules are changing before our eyes, and the optimal configuration for some of these new platforms relevant to hosting high-performance SAP solutions has yet to be determined. So minimize your deployment and upgrade risks by testing and iteratively tuning early in the game and reap the rewards.