

SECOND EDITION

Java™ Performance Tuning

Jack Shirazi

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 18

Tuning EJBs

In this chapter:

- Primary Design Guidelines
- Performance-Optimizing Design Patterns
- The Application Server
- More Suggestions for Tuning EJBs
- Case Study: The Pet Store
- Case Study: Elite.com

The performance of EJB-based J2EE systems overwhelmingly depends on their design. If you get the design right, tuning the server is similar to tuning a J2SE system: profile and tune the server, targeting object creation as a priority (since the consequences in a multiuser system are an order of magnitude greater). If you get the design wrong, you are unlikely to simply tweak your way to adequate performance. In contrast, a J2SE application can often achieve adequate performance with a non-optimal design after sufficient performance tuning. This design sensitivity is one of the reasons why J2EE design patterns have become so popular: design patterns assist everyone from novices to experienced designers in achieving adequate performance.

This design sensitivity is also the reason for the many stories about badly performing EJB projects. EJBs are a tradeoff, like most standardized APIs. In exchange for the ability to have a standard for components that developers, managers, tool vendors, and other third-party producers all work together to use, there are some overheads and design issues. Make no mistake: using EJBs compared to build-it-completely-your-way almost always incurs more overhead for your runtime system. Chapter 12 compared a proprietary communication layer to RMI (see Table 12-1), and the situation with EJBs is quite similar. Proprietary is almost always faster. It is also usually more difficult to maintain and support. EJBs have third-party support products for development, testing, tuning, deploying, scaling, persisting, clustering, and load balancing. If you don't need standardized components, EJBs may not be the best option for your project. A plain J2SE + JDBC solution has its own advantages.

Primary Design Guidelines

In “Performance-Optimizing Design Patterns” later in this chapter, I describe several design patterns that help EJB systems attain adequate performance. But first, I will discuss some primary design guidelines to consider before you can apply patterns.

Coarse-Grained EJBs Are Faster

EJBs should be designed to have large granularity—one remote invocation to an EJB should perform a large amount of work instead of requiring many remote invocations. This criterion is extremely important for a successful EJB design. Coarse-grained EJBs tend to provide a more efficient application because they minimize the number of remote communications needed to complete the work.

A more refined guideline is that any remotely accessed EJBs should be coarse-grained. Any EJBs that are *always* accessed locally can be fine-grained, if the local access is not treated as a remote access. Bear in mind that prior to the EJB 2.0 specification, *all* EJB access was (theoretically) treated remotely, even with EJBs in the same container. This means that the parameters could always be marshaled and passed through a socket, incurring a significant portion of remote-calling overhead. (Some application servers detect local EJB communication automatically and optimize that communication to avoid remote-calling overhead.) Since EJB 2.0, local entity beans can be defined, allowing optimized communications for local EJBs. But that is not a runtime decision, so it needs to be factored into the design. Local EJBs were added to the EJB specification to address this issue of improving performance among locally collocated EJBs.

The following are some detailed guidelines for achieving this combination design target of coarse-grained remote EJBs and fine-grained local EJBs. In the following list, I consider EJBs either local or remote, but an EJB can implement both interfaces, if appropriate to your application.

- Design the application to access entity beans from session beans. This optimizes the likelihood that an EJB call is local and supports several other design optimizations (listed in the subsequent section covering design patterns).
- Determine which EJBs will be collocated within the same VM. These EJBs can communicate with one another by using optimized local communications.
- Those EJBs that will (always) be collocated should be:
 - Defined as local EJBs (from EJB 2.0); or
 - Defined normally as remote EJBs and collocated within an application server that is capable of optimizing local EJB communications; or
 - Built as normal JavaBeans, and then wrapped in an EJB to provide one coarse-grained EJB (see the `CompositeEntity` pattern).
- EJBs that communicate remotely should combine methods to reduce possible remote invocations. Multiple calls frequently specify various parameters, and these parameters can be combined as a parameter object to be passed for one remote call. “Message Reduction” in Chapter 12 gives a concrete example of how to combine methods to reduce the number of remote calls required to perform an action.

- Don't design EJBs with one access method per data attribute unless they are definitely local EJBs. (More accurately, don't define data attribute accessors and updaters as remote, as they have relatively high overheads.)
- Bear in mind that any EJB service could be called remotely if you define a remote interface for it, and try to anticipate the resulting costs to the application.

EJBs Are Not Data Wrappers

EJBs should not be simple wrappers on database table rows. An EJB should be a fully fledged business object that represents and can manipulate underlying database data, applying business logic to provide appropriate refined information to callers of the EJB. If you need to access database data, but not for business-object purposes, use JDBC directly (probably from session beans) without intermediate EJB objects. EJBs can cause multiple per-row database access and updates. While this inefficiency can be justified when the EJB adds information value to the data, it is pure overhead in the absence of such business logic, and plain JDBC could be optimized much better.

Read-Only Data Is Different

Read-only data should be identified and separated from read-write data. When treating read-only data and read-only attributes of objects, a whole host of optimizations are possible. Some optimizations use design patterns, and others are available from the application server. Transactions that consist purely of read-only data are much more efficient than read-write data. Trying to decouple read-only data from read-write data *after* the application has been designed is difficult.

Stateless Session Beans Are Faster

By definition, a stateless session bean has no state. That means that all the services it provides do not depend on what it just did. So a single stateless session bean can serve one client, then another, and then come back to the first, while each client can be in a different or the same state. The stateless session bean doesn't need to worry about which client does what. The result is that one stateless bean instance can serve multiple clients, thereby decreasing the average number of resources required per client. The stateless bean pool doesn't need to grow and shrink according to the number of clients; instead, it can be optimized for the overall rate of requests.

Most application servers support pools of stateless beans. As each bean services multiple clients, the bean pool can be kept smaller, which is more optimal. To optimize the session-bean pool for your application, choose a (maximum) size that minimizes activations and passivations of beans. The container dynamically adjusts the size to optimally handle the current request rate, which may conflict with trying to choose a single size for the pool.

Stateful beans, in contrast, require one instance for each client accessing the bean. The stateful-bean pool grows and shrinks depending on the current number of clients, increasing pool overhead. If you have stateful beans, try to remove any that are finished so that fewer beans are serialized if the container needs to passivate them (see “HttpSession Versus Stateful Session Beans” in Chapter 17, which details how explicitly removing beans can improve performance).

If you have stateful beans in your design, the best technique to reduce their overhead is to convert them to stateless session beans. Primarily, this involves adding parameters that hold the extra state to the bean methods to pass the bean the current client state whenever it needs to execute. An extended example of converting a stateful bean to a stateless bean is available in Brett McLaughlin’s *Building Java Enterprise Applications, Volume I: Architecture* (O’Reilly), and online at http://www.onjava.com/pub/a/onjava/excerpt/bldgjavaent_8/index3.html. The example even shows that you can retain the stateful-bean interface while using stateless beans by using the Proxy design pattern.

If state needs to be accessible on the server, you can hold it outside session beans, for example, in an HttpSession object, or in a global cache that provides access to the state through a unique session identifier. Converting stateful session beans to stateless session beans adds extra data to the client-server transfers, but the extra data can be minimized by using identifiers and a server data store. For high-performance J2EE systems, the advantages tend to outweigh the disadvantages.

Cache JNDI Lookups

JNDI lookups, like other remote calls, are expensive. The results of JNDI lookups are also easily cached. There is even a dedicated pattern for caching EJBHome objects (the EJBHomeFactory pattern) because it is such a frequently suggested optimization.

CMP or BMP?

Should you use container-managed persistence (CMP) or bean-managed persistence (BMP)? This is one of the most frequently discussed questions about EJBs. BMP requires the developer to add code for persisting the beans. CMP leaves the job of persisting the beans up to the application server. BMP can ultimately be made faster than CMP in almost any situation, but to do so, you would probably need to build a complete generic persistency layer—in effect, your own CMP. So let’s get back to reality. (You could build a very fast, simple persistence layer, mainly raw JDBC calls, but it would not be flexible enough for the kinds of development changes constantly made in most J2EE systems. However, if speed is the top priority, this option is viable.)

BMP can be faster for any one bean. You can build in the persistency that is required by the bean, avoiding any generic overhead. That’s fine if you have five EJB types in your application. But more realistically, with tens or hundreds of EJB types, writing optimal BMP code for each EJB and *keeping* that code optimal across versions of the

application is unachievable (though again, if you can impose the required discipline in your development changes, then it *is* achievable).

With multiple beans and bean types, CMP can apply many optimizations:

- Optimal locking
- Optimistic transactions
- Efficient lazy loading
- Efficient combinations of multiple queries to the same table (i.e., multiple beans of the same type that can be handled together)
- Optimized multi-row deletion to handle deletion of beans and their dependents

I would recommend using CMP by default. However, CMP is not yet mature, which makes the judgment more complex. It may come down to which technique your development team is more comfortable with. If you do use CMP, profile the application to determine which beans cause bottlenecks from their persistency. Implement BMP for those beans. Use the Data Access Object design pattern (described later) to abstract your BMP implementations so you can take advantage of optimizations for multiple beans or database-specific features. (You may also need to use BMP where CMP cannot support the required logic—e.g., if fields use stored procedures, or one bean maps to multiple tables.)

EJB Transactions

Tuning EJB transactions is much like tuning JDBC transactions; you will find “Transaction Optimization” in Chapter 16 very relevant for EJB transactions. There are a few additional considerations. The following list summarizes optimal transaction handling for EJBs:

- Keep transactions short.
- Commit the data after the transaction completes rather than after each method call. That is, if multiple methods are executed close together, each needing to execute a transaction, then combine their transactions into one transaction. The target is to minimize the overall transaction time rather than simplistically targeting each currently defined transaction.
- Try to perform bulk updates to reduce database calls.
- For very large transactions, use the transaction attribute `TX_REQUIRED` to get all EJB method calls in a call chain to use the same transaction. Use a session façade that provides a high-level entry point so that all the methods called from that point are included in one transaction.
- Optimize read-only EJBs to use read-only transactions. Use `read-only` in the deployment descriptor to avoid unnecessary calls to `ejbStore()` by the application server (not all application servers support this feature).

- Choose the lowest-cost transaction isolation level that avoids corrupting the data. Transaction levels in order of increasing cost are `TRANSACTION_READ_UNCOMMITTED`, `TRANSACTION_READ_COMMITTED`, `TRANSACTION_REPEATABLE_READ`, and `TRANSACTION_SERIALIZABLE`.
- Don't use client-initiated transactions in the EJB environment because long-running transactions increase the likelihood of conflict, making rows inaccessible to other sessions. If the client controls the duration of the transaction, you may have no way to force the transaction to close from the server, thus allowing long or indefinite transactions. The longer a transaction lasts, the more likely it is to conflict with another transaction.
- If you need client-initiated transactions, set an appropriate transaction timeout in the *ejb-jar.xml* deployment descriptor file. Setting a timeout ensures that the application doesn't start leaking resources from transactions that are opened at the client but not completed. The deployment descriptor should be something like "trans-timeout-seconds," and you should specify a timeout that is long enough for users to reasonably complete their task.
- Declare nontransactional methods of session beans with `NotSupported` or `Never` transaction attributes (in the *ejb-jar.xml* deployment descriptor file).
- Use a dirty flag where supported by the EJB server or in a BMP or DAO implementation to avoid writing unchanged EJBs to the database. Dirty flags are a standard way to avoid writing unchanged data. The write is guarded with the dirty flag and performed only if the flag is dirty. Initially the flag is clean, and any change to the EJB sets the flag to dirty.

Performance-Optimizing Design Patterns

This is not a book on design patterns. However, because of their importance to EJB design, this section lists design patterns that are particularly relevant. You can find articles that detail these performance-optimizing design patterns at <http://www.JavaPerformanceTuning.com/tips/patterns.shtml>.

Reducing the Number of Network Trips: The Value Object Pattern

A Value Object encapsulates a set of data values. Use a Value Object to encapsulate all of a business object's data attributes and access the Value Object remotely rather than accessing individual data attributes one at a time. The Value Object sends all data in one network transfer. "Message Reduction" in Chapter 12 shows how to use the Value Object pattern to reduce the number of network transfers required to access multiple data attributes. The Value Object pattern can be used bidirectionally to improve performance—i.e., to minimize the number of network transfers to transfer data *to* the server as well as *from* the server. One variation, the Value Object Assembler pattern, uses a Session EJB to aggregate all required data from different EJBs as various types of Value Objects.

Using a Value Object may result in very large objects being transferred if too many data attributes are combined into one Value Object. A large Value Object may still be more efficient than separate multiple remote requests, but typically, only a subset of the data held by a large Value Object is needed, in which case the large Value Object should be broken down into multiple smaller Value Objects, each holding the data subset required to satisfy its remote request. This last approach minimizes both the number of network transfers and the amount of transferred data.

Once transferred, the Value Object's data is no longer necessarily up to date. So if you use the Value Object to hold the data locally for a period of time (as a locally cached object), the data could be stale and you might need to refresh it according to your application's requirements.

Optimizing Database Access: The Data Access Object Pattern and the Fast Lane Reader Pattern

Use Data Access Objects to decouple business logic from data-access logic, allowing decoupling of data-access optimizations from other types of optimizations. Data Access Objects usually perform complex JDBC operations behind a simplified interface, providing a platform for optimizing those operations. Data Access Objects allow optimizations in bulk access and update for multiple EJBs, and also allow specialized optimizations by using database-specific optimized access features while keeping complexity low.

For read-only access to a set of data that does not change rapidly, use the Fast Lane Reader pattern, which bypasses the EJBs and uses a (possibly nontransactional) Data Access Object that encapsulates access to the data. The Data Access Object in the Fast Lane Reader pattern accesses the database to get all the required read-only data efficiently, avoiding the overhead of multiple EJB accesses to the database. The resulting data is transferred to the client using a Value Object. The Value Object can also be cached on the server for repeated use, improving performance even further. This means that the Fast Lane Reader pattern efficiently reads unchanging (or slowly changing) data from the server and displays all of the data in one transfer.

Efficiently Transferring Large Datasets: The Page-by-Page Iterator Pattern and the ValueListHandler Pattern

If long lists of data are returned by queries, use the Page-by-Page Iterator pattern. This pattern is used when the result set is large and the client may not need all of the results. It consists of a server-side object that holds data on the server and supplies batches of results to the client. When the client makes a request, the results of the request are held in a stream-like object on the server, and only the first "pageful" of results is returned. The client can control the page size, and when data from the next page needs to be viewed, the whole page is sent. "Batching II" in Chapter 12 shows

how to use a Page-by-Page Iterator pattern to reduce the amount of transferred data and improve client display time.

Note that the Page-by-Page Iterator pattern actually increases the number of transfers made. However, it is an essential pattern for any server handling multiple requests that may return large amounts of data to clients. When implementing the Page-by-Page Iterator pattern, you should try to avoid making copies of the data on the server. If the underlying collection data is concurrently altered, care should be taken to ensure the client gets consistent pages. There is no upper limit to the size of a result set that this pattern can handle.

The `ValueListHandler` pattern combines the Page-by-Page Iterator pattern with the Fast Lane Reader pattern. The `ValueListHandler` pattern avoids using multiple Entity beans to access the database. Instead, it uses Data Access Objects that explicitly query the database and return the data to the client in batches rather than in one big chunk, as in the Page-by-Page Iterator pattern.

Caching Services: The Service Locator, Verified Service Locator, and EJBHomeFactory Patterns

The Service Locator pattern improves performance by caching service objects with a high lookup cost. For example, `EJBHome` objects and other JNDI lookups are often costly, but need to be performed regularly. However, many such objects are infrequently changed and thus ideal for caching. The Service Locator pattern simply interposes a Service Locator between the object initiating the lookup and the actual lookup. The Service Locator caches any looked-up object and returns the cached object where possible.

The Verified Service Locator pattern anticipates that objects in the Service Locator cache occasionally become stale and need to be refreshed. The Verified Service Locator periodically and asynchronously tests the cache elements to identify and refresh stale objects. An asynchronous periodic test minimizes the impact of stale objects to callers of the service, which would otherwise require a time-consuming synchronous call to obtain a refreshed service object. The Verified Service Locator pattern is just one variety of cache-element management among many, such as least-recently-used, element timed expiration, etc. The Verified Service Locator pattern element management is appropriate for JNDI lookups, when cache elements need to be refreshed only when the JNDI server is restarted, which should be infrequently.

The `EJBHomeFactory` pattern is simply a `ServiceLocator` dedicated to `EJBHome` objects. It is such a frequently mentioned optimization that it was given its own name.

Combining EJBs: The Session Façade and CompositeEntity Patterns

Use a Session Façade to provide a simple interface to a complex subsystem of enterprise beans and to reduce network communication requirements. The Session

Facade is normally a session bean that encapsulates the interfaces needed to work efficiently with a set of EJBs. The client communicates efficiently with the session bean, which in turn manages all the EJB calls necessary to complete the operation represented by the Session Façade. The Session Façade can communicate with the EJBs by using local calls rather than remote calls, potentially making the whole operation much more efficient. Communication between the client and the Session Façade is often best handled using Value Objects so that EJB remote interfaces are not transferred across the network. The façade can also handle security and logging more efficiently than multiple EJBs, which would each separately require security checks and logging output.

The CompositeEntity pattern reduces the number of actual entity beans by wrapping multiple Java objects (which would each otherwise be an entity bean) into one entity bean. It is used less frequently than the Session Façade pattern.

Reusing Objects: The Factory and Builder Patterns

The Factory pattern allows optimizations to occur at the object-creation stage by redirecting object-creation calls to a factory object. “Object Design” in Chapter 13 discusses this pattern.

Use the Builder pattern to break the construction of complex objects into a series of simpler Builder objects. A Director object combines the Builders to form a complex object. You can then use Recycler (a type of Director) to replace only the broken parts of the complex object, reducing the number of objects that need to be re-created.

Reducing Locking Conflicts: The Optimistic Locking Pattern

The Optimistic Locking pattern checks for data integrity only at update time and uses no locks. This feature increases the scalability of an application compared to pessimistic locking, since lock contention is avoided. The Optimistic Locking pattern is appropriate when concurrent access predominates over concurrent update (i.e., most sessions spend most of their time reading data, and very little time writing data). If sessions are transactional, transactions should be short.

Write-write conflicts with optimistic transactions can be detected using:

Timestamps

The updated row contains a timestamp field that should not be newer than when the row was accessed or the transaction started.

Version counters

A simple version counter is maintained and checked to ensure that it matches the version at transaction beginning.

State comparisons

At update time, all relevant database data is checked to ensure that it matches the “old” data.

Optimistic locking has high rollback costs when conflicts are detected, so it should not be used when conflicts are frequent.

Load Balancing: The Reactor and Front Controller Patterns

The Reactor pattern demultiplexes events and dispatches them to registered object handlers. It is similar to the Observer pattern (not described here), but the Observer handles only a single source of events, whereas the Reactor pattern handles multiple event sources. The Reactor pattern enables efficient load-balancing servers with multiplexing communications. The multiplexing of network I/O using NIO Selectors is an excellent example of the Reactor pattern. See “Multiplexing” in Chapter 8.

The Front Controller pattern centralizes incoming client requests, channeling all client requests through a single decision point that lets you balance the application at runtime (see also “Clustering and Load Balancing” in Chapter 15). This pattern also allows optimizations in aggregating the resulting view.

Optimized Message Handling: The Proxy and Decorator Patterns

Proxy and Decorator objects let you redirect, batch, multiplex, and delay method invocations. They enable application partitioning by intelligently caching data or forwarding method invocations. See “Caching” and “Application Partitioning” in Chapter 12, which show how to use proxies to improve the efficiency of a distributed application. The Proxy pattern differentiates by Proxies often instantiating their real objects, while the Decorator pattern rarely does. A Proxy object is usually created as a wrapper on the “real” object, and other objects only ever get to handle the Proxy. The Decorator is more typically given the “real” object to wrap, allowing access to both the Decorator and the “real” object. Synchronized wrappers are an example of the Decorator pattern: you can pass the original collection object to the wrapper factory and access both the original collection and the wrapped collection.

Optimizing CPU Usage: The Message Façade Pattern

The Message Façade pattern encapsulates a method call into an object that can be executed asynchronously, allowing process flow to continue without blocking. This pattern is ideal for remotely invoked methods that don’t need to return a value; remotely invoked methods that do return values can also be accommodated by storing the result for later retrieval.

The Application Server

Considerations other than performance frequently drive the choice of application server. That might not be as serious as it sounds, since all the most popular application servers target good performance as an important feature. I often read about

projects in which the application server was exchanged for an alternative, with performance cited as a reason. However, these exchanges seem to be balanced: for each project that moved from application server A to application server B, there seems to be another that moved in the reverse direction.

Nevertheless, I would still recommend that application servers be evaluated with performance and scalability as primary criteria. The ECperf benchmark may help differentiate EJB server performance within your short list of application servers. Performance-optimizing features to look for in an application server include:

Multiple caches

Application servers should offer multiple caches for session beans, EJBs, JNDI, web pages, and data access. Caching provides the biggest improvement in performance for most enterprise applications.

Load balancing

Load balancing is absolutely necessary to support clustered systems efficiently.

Clustering

Clustering is necessary for large, high-performance systems.

Fault-tolerance (hot replacement of failed components)

If one part of the system goes down, a fault-tolerant system suffers performance degradation. However, a system without fault tolerance has no service until the system is restarted.

Connection pooling

You can roll your own connection pool, but one should come standard with any application server.

Thread pooling, with multiple users per thread

Thread pooling should also be a standard feature. It is necessary to efficiently manage system resources if your application uses hundreds or thousands of threads or serves hundreds or thousands of users.

Optimized subsystems

All subsystems, including RMI, JMS, JDBC drivers, JSP tags, and cacheable page fragments, should be optimized, and the more optimized, the better. Naturally, optimized subsystems provide better performance.

Application distribution over multiple (pseudo) VMs

Distributing over VMs provides fault tolerance. The latest VMs with threaded garbage collection may not benefit from this option.

Distributed caching with synchronization

Supported directly by the application server, distributed caching with synchronization lets clustered servers handle sessions without requiring that a particular session always be handled by one particular server, enhancing load balancing.

Optimistic transaction support

Optimistic transactions reduce contention for most types of applications, enabling the application to handle more users.

Distributed transaction management

If you need distributed transactions, they are usually handled more efficiently if the application server supports them.

In-memory replication of session state information

Holding session state information in memory allows clustered servers to handle sessions without requiring that a particular session be handled by one particular server, enhancing load balancing.

No single points of failure

Eliminating single points of failure helps fault tolerance. Of course, your application may have its own single points of failure.

Hot-deploy and hot-undeploy applications for version management

You will need to upgrade your application multiple times. Hot-deployment lets you do so with almost no downtime, enhancing 24/7 availability.

Performance-monitoring API

A performance-monitoring API is useful if you need to monitor internal statistics, and an application server with a performance-monitoring API is more likely to have third-party products that can monitor it.

Performance-monitoring and analysis tools

More is always better, I say.

Security Layer

A security layer affects response times adversely. Try to dedicate separate application servers to handle secure transactions. Most types of security (SSL, password authentication, security contexts and access lists, and encryption) degrade performance significantly. Many systems use the frontend load balancer to decrypt communications before passing on requests. If using it is feasible, it is worth considering. In any case, try to consider security issues as early as possible in the design.

Gross Configuration

The gross configuration of the system might involve several different servers: application servers, web servers, and database servers. An optimal configuration runs these servers on different machines so each has its own set of specifically tuned resources. This avoids access conflicts with shared resources.

When this separation is not possible, you need to be very careful about how the servers are configured. You must try to minimize resource conflicts. Allocate separate disks, not just separate partitions, to the various servers. Make sure that the operating-system page cache is on yet another disk. Limit memory requirements so it is not possible for any one server to take an excessive amount of memory. Set process priority levels to appropriately allocate CPU availability (see Chapter 14 for more details).

When request rates increase, you should be able to maintain performance by simply adding more resources (for instance, an extra server). This target requires both a well-designed application and correctly configured application servers. Try load-testing the system at higher scales with an extra application server to see how the configuration requirements change.

Tuning Application Servers

Application servers have multiple configuration parameters, and many affect performance: cache sizes, pool sizes, queue sizes, and so on. Some configurations are optimal for read-write beans, and others are for read-only beans, etc. The popular application-server vendors now show how to performance-tune their products (see <http://www.JavaPerformanceTuning.com/tips/appservers.shtml>). Several application servers also come with optional “performance packs.” These may include performance-monitoring tools and optimal configurations, and are worth getting if possible.

The single most important tuneable parameter for an application server is the VM heap size. Chapters 2 and 3 cover this topic in detail. For long-lived server VMs, memory leaks (or, more accurately, object retention) are particularly important to eliminate. Another strategy is to distribute the application over several server VMs. This distribution spreads the garbage-collection impact, since the various VMs will most likely collect garbage at different times.

Optimal cache and pool sizing are the next set of parameters to target. Caches are optimized by trying to get a good ratio of hits to misses (i.e., when an attempt is made to access an object or data from the cache, the object or data is probably in the cache). Too small a cache can result in useful objects/data being thrown away to make way for new objects/data. Too large a cache uses up more memory than is required, taking that memory away from other parts of the system. Look at the increase in cache-hit rates as memory is increased, and when the rate of increase starts flattening out, the cache is probably at about the right size.

Each pool has its own criteria that identify when it is correctly sized. Well-sized bean pools minimize activation and passivation costs, as well as bean creation and destruction. A well-sized connection pool minimizes the amount of time requests have to wait for an available connection. If the connection pool can vary in size at runtime, the maximum and minimum sizes should minimize the creation and destruction of database connections. For thread pools, too many threads causes too much context switching; too few threads leaves the CPU underutilized and decreases response times because requests get queued.

Other parameters depend on what the application server makes available for tuning. For example, as connections come into the server, they are queued in the network stack “listen” queue. If many client connections are dropped or refused, the TCP listen queue may be too short. However, not all application servers allow you to alter the listen queue size. (See the `backlog` parameter, the second parameter of the `java.net.ServerSocket` constructor.)

More Suggestions for Tuning EJBs

A few additional tuning suggestions for EJBs are listed here:

- `Beans.instantiate()` incurs a filesystem check to create new bean instances in some application servers. You can use the Factory pattern with `new` to avoid the filesystem check.
- Tune the message-driven beans' pool size to optimize the concurrent processing of messages.
- Use initialization and finalization methods to cache bean-specific resources. Good initialization locations are `setSessionContext()`, `ejbCreate()`, `setEntityContext()`, and `setMessageDrivenContext()`; good finalization locations are `ejbRemove()` and `unsetEntityContext()`. Failures to allocate or deallocate resources need to be handled.

Case Study: The Pet Store

Sun created a J2EE tutorial application called the Pet Store.* In the Pet Store, there was no attempt to focus on performance. In a marketing coup in early 2001, Microsoft took the badly performing basic Pet Store application and reimplemented and optimized it in .NET, using the results to “show” that .NET was over 20 times faster than J2EE. (The .NET optimizations appear mostly to have been SQL optimizations together with moving much of the application server logic to database-stored procedures.) A few weeks later, Oracle took the original Pet Store code, keeping it in J2EE, and optimized the application.† The resulting optimized J2EE application performed over 20 times faster than the .NET implementation.

Here's how Oracle optimized the application:

Optimized lazy loading of data

The original Pet Store didn't try to optimize data handling. All information that might be needed is automatically loaded on the client. This is unrealistic in real-world applications that should minimize data transfers. Oracle changed the application to load only needed information. (Lazy loading is discussed in Chapters 4, 8, and 12.)

SQL query optimization

The Pet Store made no attempt to optimize the SQL queries. This lack of optimization is appropriate for a tutorial, where the most simple SQL is easier to

* See http://java.sun.com/blueprints/code/index.html#java_pet_store_demo.

† Oracle's Pet Store benchmark report is available at http://otn.oracle.com/tech/java/oc4j/pdf/9ias_net_bench.pdf. You may also want to check out the discussion of the Oracle improvements in the *Server Side* (http://www.theserverside.com/home/thread.jsp?thread_id=12753).

understand. Oracle converted some SQL to more optimal statements. (Chapter 16 discusses SQL optimization.)

No unnecessary updates

Oracle changed the EJBs so they use `isModified()` to avoid unnecessary database updates. It is always good practice to avoid doing what doesn't need to be done. (This is a good example of using a dirty flag, discussed earlier in the "EJB Transactions" section.)

Reduced contention to improve scalability

Some methods opened multiple database connections. These methods were rewritten to use only one connection at a time, reducing contention and increasing scalability. (SQL optimization is discussed in Chapter 16, and contention costs in Chapter 15.)

Limited number of items retrieved by queries

The Pet Store application default settings produced too much unnecessary data. Oracle used the Page-by-Page Iterator pattern with limited page size to improve performance and scalability.

Session data stored in session, not context

Session data was moved from the `ServletContext` to the `HttpSession`, and the JSP was modified to use the session rather than the context. Without this change, multiuser access to the Pet Store application was very limited, as all catalog access to the DB was forced through a single connection. (This topic is discussed briefly in "Cache Tags" in Chapter 17.)

Connections shortened

Connection code was rewritten to keep the DB connections very short, as is optimal with connection pooling. (SQL optimization is discussed in Chapter 16, and transaction optimization in the "EJB Transactions" section.)

String handling optimized

String-handling code was rewritten to use `StringBuffer` instead of `String`, removing unnecessary concatenations. (Chapter 5 discusses string optimizations.)

The combined effect of these optimizations from Oracle produced a greater than 400-fold improvement in performance.

Case Study: Elite.com

Elite.com is a successful Internet startup subsidiary of Elite Information Group, and provides an online time and billing solution. David Essex reviewed the J2EE technology behind the Elite.com web site for *Enterprise Development* magazine.* Elite.com's solution is similar to many J2EE implementations, leveraging the full range of J2EE

* See <http://www.devx.com/upload/free/features/entdev/2000/07jul00/de0007/de0007-1.asp>.

technologies as well as other non-Java technologies. The report covers only a few performance enhancements, but they show some of the most common high-level J2EE performance issues:

Perform work asynchronously whenever possible

Elite.com includes a queueing subsystem that asynchronously accepts external communications, such as email entries. Entries can be batched and run with minimal impact on the online system. (Chapters 12 and 15 discuss asynchronous queueing.)

Concurrency conflicts are the biggest limitation to scalability

Like most enterprise applications, Elite.com experienced conflicting concurrent access to some resources. When this caused severe decreases in performance in one subsystem, Elite.com solved the problem by using a resource pool (an EJB connection pool shared among servlets), improving the subsystem performance. (Contention costs are discussed in Chapter 15.)

Local is much faster than remote

Moving components so they are local to each other can significantly improve performance by eliminating marshalling and remote-transfer overhead. Collocating the EJBs and servlets and converting the communication to local calls can speed performance dramatically. (This topic was discussed earlier in this chapter.)

Performance Checklist

- The performance of EJB-based J2EE systems overwhelmingly depends on getting the design right. Use performance-optimizing design patterns: Value Object, Page-by-Page Iterator, ValueListHandler, Data Access Object, Fast Lane Reader, Service Locator, Verified Service Locator, EJBHomeFactory, Session Façade, CompositeEntity, Factory, Builder, Director, Recycler, Optimistic Locking, Reactor, Front Controller, Proxy, Decorator, and Message Façade.
- Explicitly remove beans from the container when a session is expired. Leaving beans too long will get them serialized by the container, which can dramatically decrease performance.
- Coarse-grained EJBs are faster. Remote EJB calls should be combined to reduce the required remote invocations.
- Design the application to access entity beans from session beans.
- Collocated EJBs should be defined as Local EJBs (from EJB 2.0), collocated within an application server that can optimize local EJB communications, or built as normal JavaBeans and then wrapped in an EJB to provide one coarse-grained EJB (CompositeEntity design pattern).
- EJBs should not be simple wrappers on database data rows; they should have business logic. To simply access data, use JDBC directly.

- Stateless session beans are faster than stateful session beans. If you have stateful beans in your design, convert them to stateless session beans by adding parameters that hold the extra state to the bean methods.
- Optimize read-only EJBs to use their own design, their own application server, read-only transactions, and their own optimal configuration.
- Cache JNDI lookups.
- Use container-managed persistence (CMP) by default. Profile the application to determine which beans cause bottlenecks from their persistency, and implement bean-managed persistence (BMP) for those beans.
- Use the Data Access Object design pattern to abstract your BMP implementations so you can take advantage of optimizations possible when dealing with multiple beans or database-specific features.
- Minimize the time spent in any transaction, but don't shorten transactions so much that you are unnecessarily increasing the total number of transactions. Combine transactions that are close in time to minimize overall transaction time. This may require controlling the transaction manually (i.e., turning off auto-commit for JDBC transactions or using `TX_REQUIRED` for EJBs).
- J2EE transactions are defined with several isolation modes. Choose the lowest-cost transaction isolation level that avoids corrupting the data. Transaction levels in order of increasing cost are: `TRANSACTION_READ_UNCOMMITTED`, `TRANSACTION_READ_COMMITTED`, `TRANSACTION_REPEATABLE_READ`, and `TRANSACTION_SERIALIZABLE`.
- Don't leave transactions open, relying on the user to close them. There will inevitably be times when the user does not close the transaction, and the consequent long transaction will decrease the performance of the system significantly.
- Bulk or batch updates are usually more efficiently performed in larger transactions.
- Lock only where the design absolutely requires it.
- `Beans.instantiate()` incurs a filesystem check to create new bean instances in some application servers. Use the Factory pattern with `new` to avoid the filesystem check.
- Tune the message-driven beans' pool size to optimize the concurrent processing of messages.
- Use initialization and finalization methods to cache bean-specific resources. Good initialization locations are `setSessionContext()`, `ejbCreate()`, `setEntityContext()`, and `setMessageDrivenContext()`; good finalization locations are `ejbRemove()` and `unsetEntityContext()`.
- Tune the application server's JVM heap, pool sizes, and cache sizes.