MICHAEL KEITH AND RANDY STAFFORD, ORACLE

In the early 1990s, when object-oriented languages emerged into the mainstream of software development, a noticeable surge in productivity occurred as developers saw new and better ways to create software programs. Although the new and efficient object programming paradigm was hailed and accepted by a growing number of organizations, relational database management systems remained

the preferred technology for managing enterprise data. Thus was born ORM (object-relational mapping), out of necessity, and the complex challenge of saving the persistent state of an object environment in a relational database subsequently became known as the object-relational impedance mismatch.

Complex problems sometimes demand complex solutions, and ORM software is no exception. It is necessarily

Exposing the ORM Cache

Familiarity with
ORM caching issues
can help prevent
performance problems and bugs.

intricate, with multiple facets and components to handle everything from query generation and database write optimizations, to the management of object identity in the virtual machine. The average developer, simply trying to get an application working, may choose to ignore certain complexities of ORM subsystems and their configuration. Understanding fundamental features such as caching, which may be seen as a mere *optimization*, is

critically necessary to intentional and correct application design and should not be overlooked.

Caching is generally recognized as being vital to performance optimization. Studies in virtually every computational domain have shown that caching can enhance performance and increase throughput,¹ and rarely will any such claims evoke even a hint of debate. Failure by developers to understand an ORM product's caching

approach, however, can produce anomalous application behavior, unexpected results, or outright bugs. User forums are littered with evidence of developers suffering the consequences of such failures of understanding.

Caching can be one of the most technologically advanced components of an ORM implementation, thus representing a critical balance point for any application that uses the implementation. Failure to acknowledge it as a potential fulcrum may result in an application teetering or falling on the side of poor performance and incorrect semantics. In this article, therefore, we discuss topics relevant to caching in ORM systems, and we expose some of the details that implementations must be concerned with and that application developers should be aware of.

OBJECTS AND IDENTITY

First and foremost developers must acknowledge the nature of objects and how they are used in object-oriented languages. In practice, very rarely does an object exist in isolation from other objects. An application reference to an object is really an indirect reference to an entire graph of objects rather than to a single solitary object. The consequences of such a realization are far-reaching and form the basis for many of the difficulties associated with caching in ORM.

When a read operation is performed, it must be considered by runtime that the process may also fault in objects referenced by the asked-for object. Of course, this sequence may continue recursively, causing a whole multitude of objects to be read from the database, each individually requested as needed and in succession (a phenomenon dubbed *ripple loading*²). Developers can prevent this from happening through one of many backstop measures, such as declaring, either statically or dynamically, whether specific relationships should be traversed and loaded. There are other approaches to avoiding multiple successive trips to the database, but a discussion of these is outside the scope of this article.

An object graph, by definition, implies that there may be multiple paths leading to the same object. In some cases these multiple relationships may be from a single object, but in most cases they are from different objects. In the course of loading the object graph, these relationships must end up pointing to the same identical object, not two distinct memory imprints that happen to have the same state. Failure to maintain object identity will lead to the persistent state of the object being duplicated in multiple instances, each one containing a point-in-time view of the entity state. This will inevitably lead to inconsistent state and incorrect program behavior.

Maintaining the identity of objects in a graph means that the loader must keep track of each object and its identity. The nature of the solution meshes neatly with the job that a cache already has to do, so it is not surprising that the task is often relegated to the cache.

CACHING LEVELS

An application manages different visibility scopes during its execution. For single-user scopes an isolated cache is appropriate, but for global contexts a shared cache, sometimes referred to as an L2 (level 2) cache, provides the level of caching that offers the same state to all requesters. Each of these is unique to its purpose and may function or perform slightly differently from the other. There may even be duplication of state spanning the two caches, particularly in light of isolation requirements.

TRANSACTIONAL CACHE

Transactions clearly play a major role in any system, including the cache. In fact, the transactional cache is purposed especially for the transaction, and its inhabitants are strictly transactional objects. Being associated with the transaction implies that the cache exports the correct isolation and consistency of its objects (the “correct” isolation is described in more detail in a later section). Assumptions about the type of transaction are particularly relevant because of the differences among them. Some are thread-bound, while others allow multi-threading; some are tied to a single database connection, while others may access multiple resources.

The presence of an object in the transactional cache means, by definition, that it is transactional. There is an if-and-only-if relationship between the two, such that when a transactional object is modified, its modified state must be reflected within the transactional cache. Furthermore, the state of the transactional cache represents the total change summary of the transaction from the ORM perspective and must of necessity follow the life cycle

of the transaction. If the transaction gets rolled back, then the changes will be discarded; but if the transaction commits, then the logical change summary contained in the cache will be converted to SQL and committed to the database.

SHARED CACHE

The vast majority of operations in most applications are read operations. When there are no plans to modify the object state, either within or outside a transaction, then a globally shared cache is the most efficient mechanism for obtaining the read-only state, even if it is deemed read-only for a temporary period.

A globally shared cache is accessible by all clients in the same process space, whether or not they are in a transaction context. Some implementations even allow read-only operations to occur within a transaction, returning nontransactional shared state instead of transactional state. The motivation for doing so is usually performance, since there is a nonzero cost to making an object transactional. The consequence of not following the rules, however, could be severe. If the caller modifies an object, previously assumed to be read-only, then its changes will not be reflected in the database and the cached object will end up in a corrupt and inconsistent state, containing uncommitted updates outside the transaction.

INTERCACHE INTERACTION

Even though transactional caches do not outlive the transaction, retaining the changes contained in those transactional caches is much more efficient. The sum total of the object changes needs to be reflected in the shared cache so that it can be kept up to date. If this merge step were not done, then the versions of the objects in the shared cache would be outdated or stale, necessitating extra refresh operations from the database. In the case of updates, then, the direction of update data is from the transactional cache to the shared cache.

From the read perspective it works the opposite way. When an object becomes transactional, the most recent known state of the object can normally be obtained from the shared cache; hence, the transactional cache acts as a consumer of the shared cache in order to save itself a trip to the database. Figure 1 is a diagrammatic view of the exchange between the caches. Note that if the shared cache cannot provide the object, then a database query must be issued to obtain it, and the resulting object can then be made available in the shared cache.

CACHING GRANULARITY

A general cache may accommodate one or multiple types of cached structures, and the same holds true for ORM caches. Although we have been referring to the ORM

cache as an object cache, in fact implementations are fairly diverse and vary in the way they store, access, and update the data contained in them. These distinctions, and the various configurations that are associated with each of them, may have different effects on performance.

OBJECT CACHE

In an object-oriented environment, the choice of what to cache tends toward the most intuitive format—that of the domain object itself. This is further supported by the realization that domain objects are what will be returned to the user eventually, anyway, and that caching in an intermediate form may introduce additional overhead each time the object must be constructed.

Transactional caches have a tendency to be exclusively object caches. Storing the objects in their native domain form is the most efficient way for the in-transaction operations to function, allowing for simple relationship traversal.

The cost of caching domain objects is that the objects must be built and preloaded with the object state at the point of reading from the database. When caching objects, there is not typically any other kind of caching, so retrieved data must be stored as part of the object aggregate. It works both ways, of course, as the benefits of refreshing or returning read-only data become more pronounced because the objects are prebuilt, thus avoiding the cost of rebuilding.

DATA CACHE

If object caching is at one end of the caching spectrum, then data caching is at the other. Caching at the data level means simply that the raw compositional state of each object is stored separately in the cache without an encapsulating object. Simple data fragments are easily manipulated and stored, with little or no accrued costs owing to object management and relationships.

One of the other main advantages of caching state in

its raw or primitive form is that it is closer to the kind of data that is being transferred to and from the basic database connectivity layer. This provides a simpler interface for exchange and renders the cache more pluggable.

The performance cost of caching data is that every successful request requires at least one—and usually more—object construction. The newly constructed objects are then hydrated from the cached data and returned to the ORM manager.

QUERIES AND CACHING

The primary motive for ORM caching is to increase performance through localized data access as an alternative to making a database round trip to retrieve it. The initial operation is always going to be the execution of a find or query call to obtain the entity or set of resulting objects; thus, caching and the queries that request objects are closely connected.

An ORM product is presumed to be on fairly familiar terms with the database it communicates with. The ORM system is not, itself, a database, however, and is not normally expected to perform queries in memory, although some do indeed support a subset of that functionality (sometimes referred to as in-memory querying). If the query criteria are based upon one or more primary-key values, or the keys against which the cached entities are stored, then the query can be satisfied by the in-memory cache. This is the optimal query-processing scenario since it avoids having to make a database round trip.

If the search criteria rely upon non-key fields, then normally the query must be executed against the database to obtain the set of result identifiers. That set can then be used to obtain the set of entities from the cache.

The trade-offs can be more clearly evaluated at this point. The data obtained from the database can be the complete set of entity data, or it can be just the identifiers. On the one hand, if the entity for a given identifier turns out not to be cache-resident, then an additional trip to the database must be undertaken to obtain the missing entity data. On the other hand, if the entity data is pulled from the database and the entity did in fact reside in cache, then the carrying cost of the retrieved data was apparently wasted. It turns out that even if the entity were cached, its contents could have become stale since the time it was loaded. In this situation the returned data can be used to refresh the cached copy with the fresh data from the database.

There is an additional mitigating factor to retrieving the entity state from the database: if the record is not large, then the cost of getting the entire record is only

fractionally greater than that of retrieving just the identifier once all the database overhead of record location is calculated.

CACHE REFERENCES AND EVICTION

Developers of Java and other object-oriented languages are very familiar with the way that garbage collectors work in the virtual machine. Objects that are no longer referenced by *live objects*—those associated with an active execution context—become garbage, and the memory they occupy is reclaimed for reuse.

One of the primary duties of a shared cache is to hold on to state that is no longer referenced by live objects, thereby preventing it from being garbage-collected. In other cases, the cache should be configured to let go of objects that are no longer needed. Ideally, a cache would know exactly when an object will no longer be needed or if it will be accessed in the near term and should be kept around. Unfortunately, a cache cannot be expected to predict the future, and it falls to the user to configure how the cache references objects based on what the user knows about the access patterns of the application. Adaptive strategies do exist where caches attempt to be “intelligent” and adapt the caching strategy based on previously observed access patterns, but these strategies are beyond the scope of this article.

The way that a cache references its cached state is typically highly configurable. The parameters are based on the conventional memory-management concepts of soft and weak referencing. (We are discussing traditional ORM, not realtime systems that must impose strict control over the number of instances and garbage-collection periods that occur.) Recall that weak references are those that point to objects that the garbage collector may reclaim if no other regular or hard references are pointing to them. Soft references are those that point to objects that can be reclaimed if the virtual machine *really* needs more heap space (and there are no hard references to the objects). Combining the two reference types in the same cache and migrating references from one type to the other can offer a dynamic balance that adjusts to the needs of both the application and the virtual machine, but gives preferential treatment to the application.

Cache eviction policies also vary, with options that include time-to-live settings that cause objects to be evicted after a specific period of time, schedules that trigger eviction at a specific day or time of day, and freshness guarantees that keep track of when objects were last accessed and evict them if the time between accesses was too great.

A sample cache reference configuration with a scheduled eviction policy is shown in figure 2. In this example, a portion of the L2 cache is reserved for softly referencing objects, leaving the rest for weak referencing. The most commonly accessed weakly referenced objects will be tenured and softly referenced.

The requirements of the application determine the size of the soft component. An appropriate balance will keep the objects that are used frequently but not always hard referenced in the soft part of the cache, without allocating an excess amount of space for unreferenced objects. The trade-off is that the cache will never cause the VM to run out of memory, but if you end up spending too much time on the fringe, then cache references may be repeatedly discarded.

By way of eviction policy, in the example in figure 2, all instances of a particular domain class are scheduled to be evicted each day at 3 a.m. This would allow the results of an overnight batch-update process to be visible the following day, regardless of cache contents and usage.

CLUSTERED CACHES

Scaling a successful ORM-based application can be significantly more difficult than its initial development, because frequently the application has not been architected *a priori* to accommodate future scaling. It is usually a myth that a functioning ORM application, running on a single server, can be scaled up unchanged by simply procuring an entire cluster of servers and running on that. In a typical ORM application the cache may be an important reason for good application performance. When there is the possibility of other processes updating the underlying database, then the individual process caches must be considered, and the overall health of the combined clustered caches must be taken into account.

The problem is that the likelihood of stale-data syn-

drome increases dramatically with each new server that operates on the same data set. Every operation that causes data mutation in the primary data source (the database) also produces the consequence that every cached version of that entity in the cluster (except for the entry cached in the server that made the update) becomes invalid. Furthermore, each of the caches needs to know, or at least have the ability to figure out, that its cache entry for that entity is stale.

A number of tactics can be used to remedy the clustered-cache problem, but most can be categorized or subsumed by one of three strategies:

- The initiating cache (the one that is doing the mutating) sends messages to the other caches in the cluster to indicate the objects that have changed. These messages may take on any form and use various transports and group communication schemes.
- Each cache tries to figure things out on its own, continually checking to see if objects have changed in the primary database and refreshing the entities in memory when necessary.
- Caches rely upon an external process to notify them of changes in the database. A cache may react to these messages simply by evicting the objects that it knows have changed, or it may go and eagerly refresh its copy from the database.

The best fit for a particular application is going to depend both upon the application itself, as well as the ability of its environment to support a given strategy. All will clearly perform better if the number of writes is sufficiently low than if it is high, since data mutation is the source of cache incoherency and the cause of traffic to render the cache coherent.

The first and third approaches would appear to be more network intensive in the face of a growing network, since adding n instances (or nodes) to a network is going to cause n additional messages to be sent (by either the initiating node or the notifier) every time an object is

changed. Even though the second approach does not send any messages to other nodes, it may have to check frequently with the database. It never knows if an object has changed, so even just to do a read it must ask the database, the source of truth, to be sure it has the most recent state. If every node is following this same procedure, then the traffic could end up being higher than the other two approaches, creating a database bottleneck and essentially executing without any caching at all.

It may be that some of the objects are immutable or that the tolerance for stale data is higher for some objects than for others, so the database needs to be consulted only for a select smaller group of objects or at a specific frequency. This might make the second approach more palatable. It might also be that only a small percentage of the objects are ever modified or that the environment doesn't allow for a connection to be established from an external database-monitoring process to the ORM system; thus, the first approach would be well suited for the task.

TRANSACTION ISOLATION

The notion that you would even need to consider the transaction isolation of a cache is foreign to some. The assumption is that a cache will work, and the isolation will be correct. The fault with this way of thinking is that there are as many different strategies of managing, loading, merging, evicting, and consulting the caches as there are products that use caching, and each of these factors may have an effect on transaction isolation.

The isolation level typically expected, whether in ignorance or by experience, from a cache is usually **READ_COMMITTED**. At the low end this is reasonable since, in general, nobody anticipates getting a query result that includes an uncommitted change from another transaction. Most serious products, therefore, do not merge the contents of their transactional caches into their shared cache until the transaction has successfully committed.

At the upper end of the spectrum there is some variety in the isolation inherent in a cache. The difference between **READ_COMMITTED** and **REPEATABLE_READ** is well defined, yet the cost is not bounded. One vendor may decide that cache safety is paramount and comes only through a given isolation level, gating all cache access by coarse-grained locks, or acquiring the locks eagerly. The problem with this well-intentioned perspective is that a practice such as serializing all cache access carries a non-trivial cost. Many applications don't have data dependencies spread across their domain model, and thus do not actually have such strict isolation requirements, but are still forced to pay the performance price.

READ_COMMITTED isolation can be sufficient and satisfactory for most applications, but those that do require stricter isolation should be permitted to perform large-scale compound cache operations atomically when necessary. The difference is that it should be a caching *option* rather than a characteristic entrenched in the implementation.

ISOLATION STRATEGIES

We just discussed transaction isolation requirements from the user perspective, but only waved our hands about the fact that such isolation requirements may be implemented differently. In this section we describe some of the common strategies for handling isolation between applications.

IMPLICIT COPY-ON-READ

One of the tried-and-true approaches to guaranteeing data consistency and complete isolation is *implicit copy-on-read*, which creates a local transactional cache copy of the object as soon as it is read in the unit of work. This certainly guarantees that whatever happens, no other application will see the changes to that object until they are written to the persistent store or merged into the shared cache. The following is a sample sequence of events in a traditional implicit copy-on-read scenario:

1. Begin tx.
2. Read object (get object from shared cache or from database).
3. Copy object and insert in tx cache.
4. Return copy to user.
5. User modifies copy.
6. Tx commit begins.
7. Modified copy contents get sent to persistent storage.
8. Tx commit completes.
9. Changes made to object are merged into shared cache.

A window exists between the time the transaction completes its commit phase and the time the changes are merged into the shared cache. This means that there is some nonzero amount of time in which another application could get a stale copy of the object from the shared cache, even though it has been updated in the database.

This window is of no real consequence, however, since if the second application is only reading the object, then it might just as well have read it before the first application committed its transaction and gotten the same stale data. The fact that it happened to have read it within that window of time has no relevance. If it were to have a stale copy of the object and perform a write on it, however, it would be bad if the changes of the first application ended

up getting overwritten or lost as a result of the staleness of the initial state of the object being written to by the second application. The solution to this problem lies in optimistic locking of the entity to ensure that no changes get overwritten or lost.³

Because the implementation automatically performs the copying without the user needing to do anything, one of the advantages of an implicit copy-on-read is that the user need not take any special action when deciding to update the object. The user can rely on the copy that it has been using to read from and perform the writes on that copy. Any and all updates will be sent to the database at the appropriate time.

A marked problem with eagerly copying-on-read is the accumulation of objects in the transactional cache. The cache does not necessarily distinguish between objects that were read and those that were updated or made transactional because an update was forthcoming. Applications that start a transaction and do a great deal of reading but only a little writing will see their transactional caches grow to include all of the objects, not just those that contain changes and need to be written out.

IMPLICIT COPY-ON-WRITE

A more efficient approach to managing the transactional cache space is to do no copying of objects as they are read into the transaction, but only as they are modified by the user. This *implicit copy-on-write* approach limits the transaction to containing only those objects that have been changed, and it does not leave the transaction vulnerable to bulging instance counts and management costs. A typical implicit copy-on-write sequence is:

1. Begin tx.
2. Read object (get object from shared cache or from database).
3. Return object to user.
4. User modifies object, causing copy to be created with changes stored inside it.
5. Insert copy in tx cache.
6. Tx commit begins.
7. Modified copy contents get sent to persistent storage.
8. Tx commit completes.
9. Changes made to object are merged into shared cache.

This appears to be an elegant approach to managing transactional data, since only dirty objects end up being copied, and this would again be performed automatically by the implementation. Objects that became transactional solely for reading turn out not really to be transactional at all and don't take up transactional space.

The catch to this strategy shows up when the copy

must be performed, or specifically when the user changes the object and the implementation recognizes that the copy must be made. An extension of that difficulty is that the object changes cannot be made directly to the shared object, but must instead be stored in the modifiable copy. This presents the implementation with the burden of having to return the changed state of the object when accessed within the transaction while keeping those changes isolated from other threads that may be accessing the shared object.

The most common technique for overcoming this challenge is to weave the class at load time so that the implementation can insert some proprietary handling code into each and every instance that gets created. This may or may not be palatable to some applications, but for most modern-day systems that use aspect-oriented programming and numerous tools and libraries that exploit the ability to perform bytecode weaving of some kind, this does not end up being an onerous restriction. There are, nevertheless, some obstacles that may still prevent certain applications from using products that rely on these kinds of techniques.

EXPLICIT COPY-BEFORE-WRITE

A third tactic to providing isolation in the face of transactional writes to objects is a twist on the previous copy-on-write, but balances the need to weave the object classes. This approach is called *explicit copy-before-write*, precisely because it requires the user to call into the ORM transactional manager to cause a copy to be made of a given object. It benefits from the advantage offered by implicit copy-on-write, in that only objects that are modified, or explicitly copied, become transactional. A sample explicit copy-before-write sequence would be:

1. Begin tx.
2. Read object (get object from shared cache or from database).
3. Return object to user.

4. User registers object copy in transaction (causing copy to be inserted in tx cache).
5. User modifies object copy.
6. Tx commit begins.
7. Modified copy contents get sent to persistent storage.
8. Tx commit completes.
9. Changes made to object are merged into shared cache.

This example makes clear that the downside is that users must either know up-front that they plan on modifying an object, or if they discover at some point in the transaction that they need to modify a specific object, then they must call for a copy to be made and begin with a new version/instance of that object. In fact, it may have even changed since the time it was previously read, so the timing of when the copy is obtained can be important.

REFERENCE TIGHTENING

We can reduce the transactional-cache footprint by incorporating the soft and weak referencing techniques (described earlier in the context of the shared cache) with the copying policies shown here. If we start with weakly referenced objects, then as they become unused or unreferenced they gradually become garbage-collected and leave the cache over the course of longer-running transactions.

The problem with weak objects in a transactional cache, though, is that we run the risk of changing an object and then moving on to another object, leaving behind the one that we already changed. If it is weakly referenced in the cache, then that object may fall out of the cache and the changes will never get written out to the backing database store. To prevent this from happening, the cache can tighten or strengthen the reference from being weak to being hard at the point the change is made. This will cause all objects containing changes to be retained for the duration of the transaction, so no changes will be lost, but unchanged objects that become dereferenced will be permitted to fall out.

UNDERSTANDING THE OPTIONS

Caching in any system can be as simple or as complex as the creator wants it to be, but the very nature of ORM tools lends to them being a little more complex than the average caching layer.

Despite the multitude of caching details that affect the performance and semantics of the ORM runtime, the goal does not need to be that every detail and nuance of the implementation be known and understood. Having a basic familiarity with the main issues and some of the possible implementation choices provides the first line

of defense against configuration errors that could lead to performance degradation or bugs.

Some of the implementation strategies discussed here are focused on optimization and performance, whereas others are geared more toward ease of use. Some ORM products have locked themselves into one particular scheme, whereas others supply multiple schemes and offer choices, providing experienced users with opportunities to benefit from the performance options that may be optimal for their applications. The better you understand the options, the better qualified you will be to build successful, performant ORM-based applications. Q

REFERENCES

1. Fernandez, J., Fernandez, A., Pazos, J. 2005. Optimizing Web services performance using caching. International Conference on Next Generation Web Services Practices.
2. Fowler, M. 2004. *Patterns of Enterprise Application Architecture*. Addison-Wesley.
3. Keith, M., Schincariol, M. 2006. *Pro EJB 3: Java Persistence API*. Apress.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MIKE KEITH has more than 15 years of teaching, research, and practical experience in distributed systems and object persistence. He sits on a number of industry specification expert groups and was the co-specification lead of the 1.0 version of JPA (Java Persistence API). He holds a master's degree in computer science from Carleton University, where he also spent time as a lecturer. He has spoken at numerous conferences worldwide, written several papers and articles for industry magazines and journals, and is coauthor of *Pro EJB 3: Java Persistence API* (Apress, 2006). He lives in Ottawa, Canada, and is employed by Oracle as a persistence and server architect.

RANDY STAFFORD has 20 years of experience as a developer, analyst, architect, manager, consultant, and author. He currently works for Oracle's middleware development organization, where he engages globally for proof-of-concept projects, architecture reviews, and production crises with diverse customer organizations, specializing in grid, SOA, performance, HA, and JEE/ORM work. He was a contributor to Martin Fowler's *Patterns of Enterprise Application Architecture* (Addison-Wesley, 2002) and Floyd Marinescu's *EJB Design Patterns* (Wiley, 2002). He lives in Denver, Colorado, with his wife and family.

© 2008 ACM 1542-7730/08/0500 \$5.00