



the
POWER
of
JAVA™

 **TANGOSOL™**



JavaOne
Hot! and Powerful. An Essential Conference.

Distributed Caching: Essential Lessons

Cameron Purdy

President Tangosol

www.tangosol.com

TS-1402

Overall Presentation Goal

Learn all about distributed caching, different ways to organize data in distributed environments, and some good rules of thumb

Speaker's Qualifications

- Cameron Purdy is President of Tangosol, and is a contributor to Java™ technology and XML specifications
- Tangosol is the Java Temporary Caching API (JSR 107) specification lead and a member of the Work Manager for Application Servers (JSR 237) expert group
- Tangosol Coherence is the leading clustered caching and data grid product for Java and J2EE™ programming environments
- Coherence enables highly scalable in-memory data management and caching for clustered Java based applications

Why Cache?

There is no better way to increase the scalable performance of applications than to use caching to unload deeper tiers!



the
POWER
of
JAVA™



JavaOne
THE JAVAAUGUST AND SEPTEMBER CONFERENCE

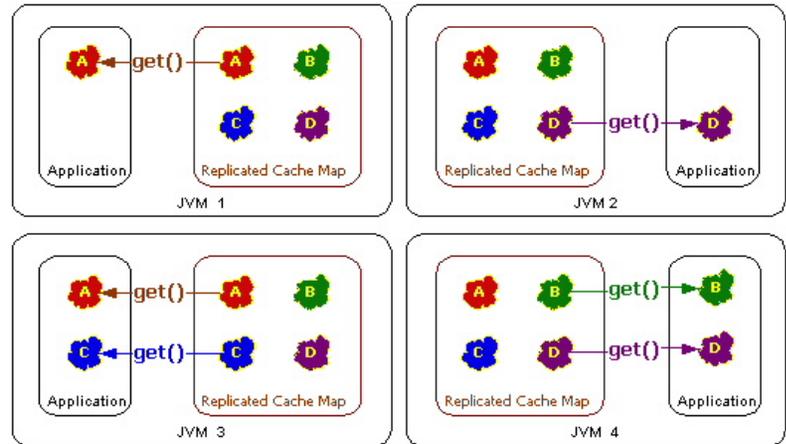
Caching Topologies

<code>/>

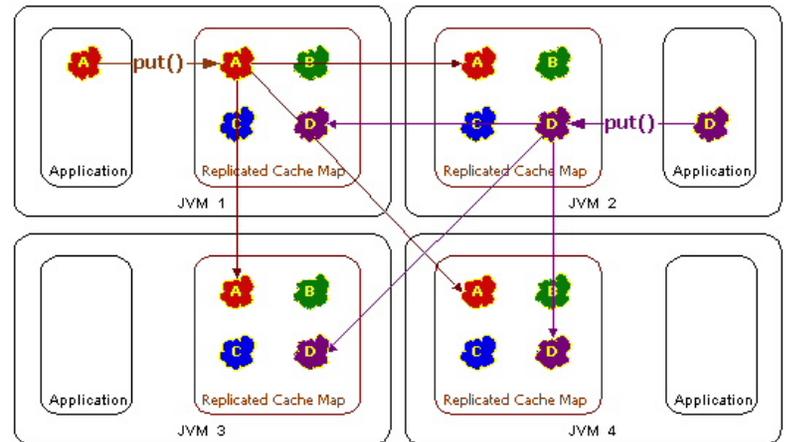
Replicated Topology

- **Goal:** Extreme Performance
- **Solution:** Cache Data is Replicated to all members of the cluster
- **Zero Latency Access:** Since the data is replicated to each cluster member, it is available for use without any waiting; this provides the highest possible speed for data access; each member accesses the data from its own memory
- **Limitations**
 - **Cost Per Update:** Updating a replicated cache requires pushing the new version of the data to all other cluster members, which will limit scalability if there are a high frequency of updates per member
 - **Cost Per Entry:** The data is replicated to every cluster member, so Java heap space is used on each member, which will impact performance for large caches

Reads



Writes



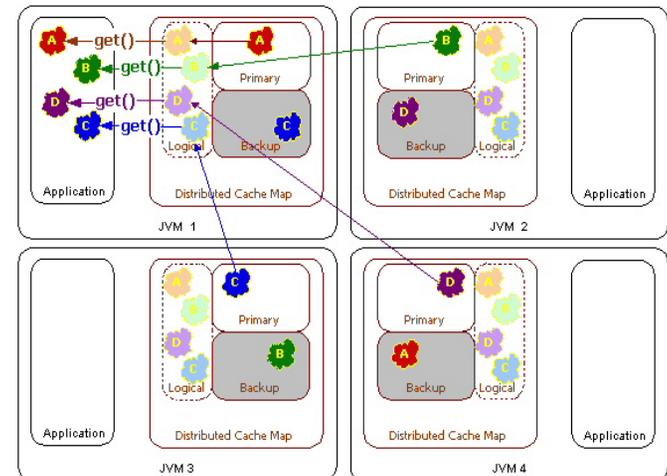
Replicated Topology: Summary

- **Performance:** Very good read performance
- **Scalability:** The scalability of replication is inversely proportional to the number of members, the frequency of updates per member, and the size of the updates
- **Uses:** Small read-intensive caches that benefit from a data “push” model

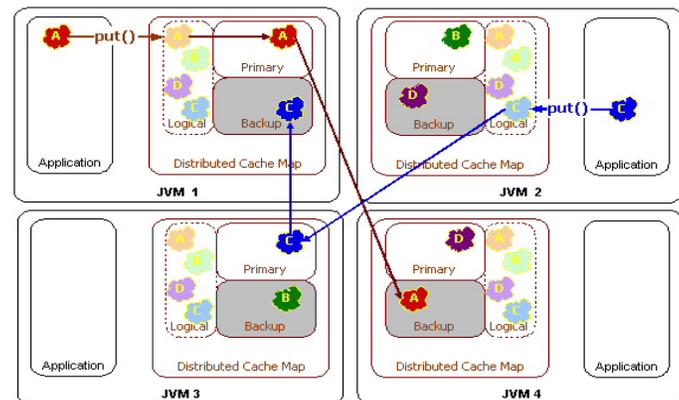
Partitioned Topology

- **Goal:** Extreme Scalability
- **Solution:** Transparently partition the Cache Data to distribute the load across all cluster members
- **Linear Scalability:** By partitioning the data evenly, the per-port throughput (the amount of work being performed by each server) remains constant
- **Benefits**
 - **Partitioned:** The size of the cache and the processing power available grow linearly with the size of the cluster
 - **Load-Balanced:** The responsibility for managing the data is automatically load-balanced across the cluster
 - **Ownership:** Exactly one node in the cluster is responsible for each piece of data in the cache
 - **Point-To-Point:** The communication for the distributed cache is all point-to-point, enabling linear scalability

Reads

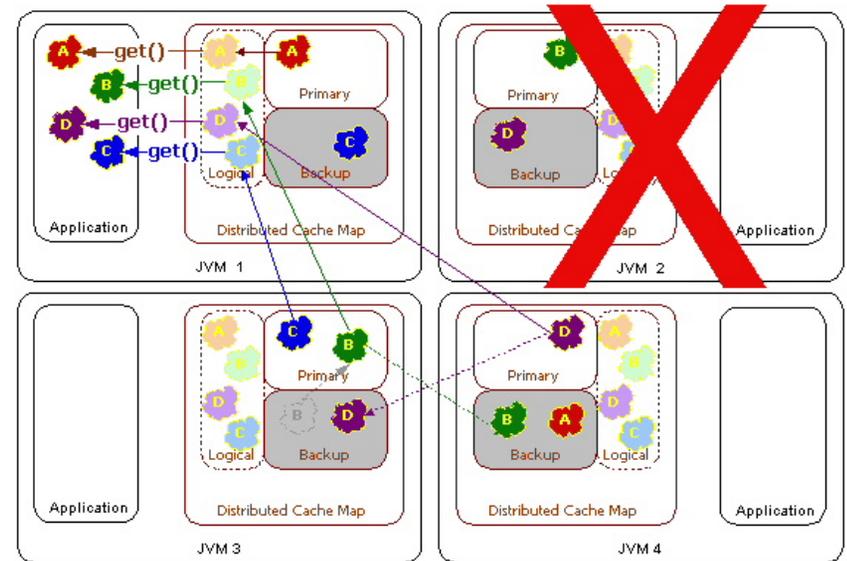


Writes



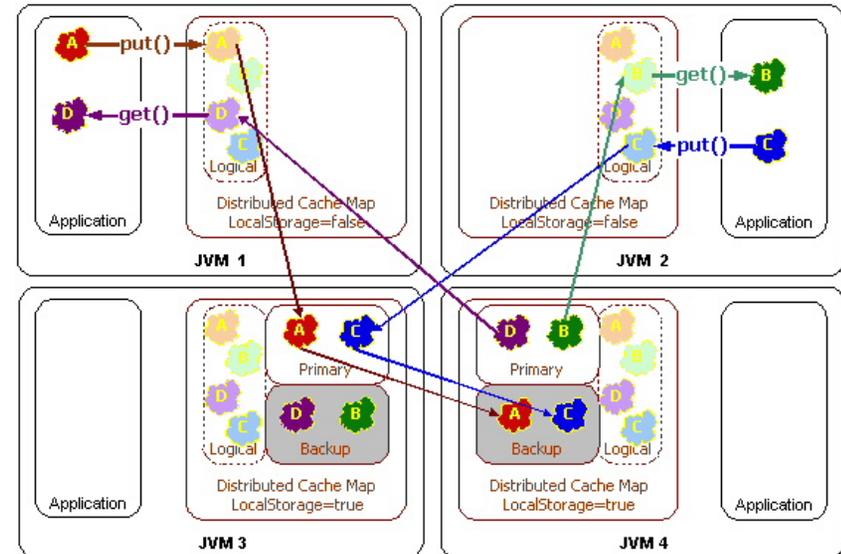
Partitioned Topology: Failover

- **Failover:** All cache services must provide lossless failover and failback
 - Configurable level of redundancy
 - Data is explicitly backed up on different physical servers (mesh architecture)
 - There is never a moment when the cluster is not ready for any server to die; no data vulnerabilities, no SPOFs



Partitioned Cache Servers

- Location Transparency:**
 The Java Temporary Caching API and its behavior are the same with a local, replicated or partitioned cache
- Local Storage Enabled:**
 Cluster nodes with local storage enabled will provide the cache and backup storage for the partitioned cache; all cluster nodes will have the same exact view of the data, because of Location Transparency

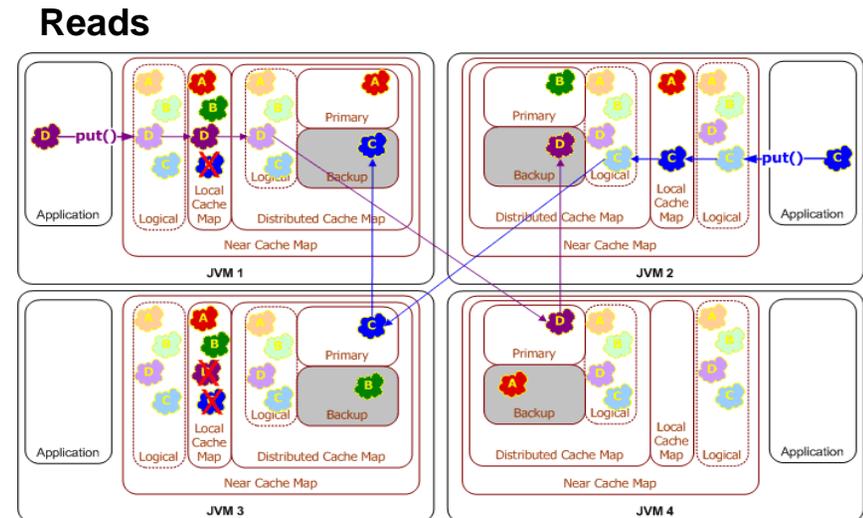


Partitioned Topology: Summary

- **Performance:** Fixed cost
- **Scalability:** Linear scalability of both cache capacity and throughput as the number of members increases; designed for scaling on modern switched networks
- **Uses:** Any size caches, scaling with the size of the cluster or data grid; both read- and write- intensive use cases; ability to offload heap usage to other Java VMs; load-balancing; resilient to server failure

Near Cache Topology

- **Goal:** Extreme Performance, Extreme Scalability
- **Solution:** Local “L1” In-Memory cache in front of a Clustered “L2” Partitioned Cache
- **Result:** Zero Latency Access to recently-used and frequently-used data; scalable cache capacity and cache throughput, with a fixed cost for worst-case



Near Topology: Summary

- **Performance:** Zero-latency for common data; fixed cost for the remainder of the data
- **Scalability:** Linear scalability of both cache capacity and throughput as the number of members increases; slightly less with Seppuku
- **Uses:** Any size caches; great for read-intensive caches with tight data access patterns; killer “Cache Server” configuration

Cache-Aside Architecture

- Cache-Aside refers to an architecture in which the application developer manages the caching of data from a data source
- Adding cache-aside to an existing application
 - Check the cache before reading from the data source
 - Put data into the cache after reading from the data source
 - Evict or update the cache when updating the data source

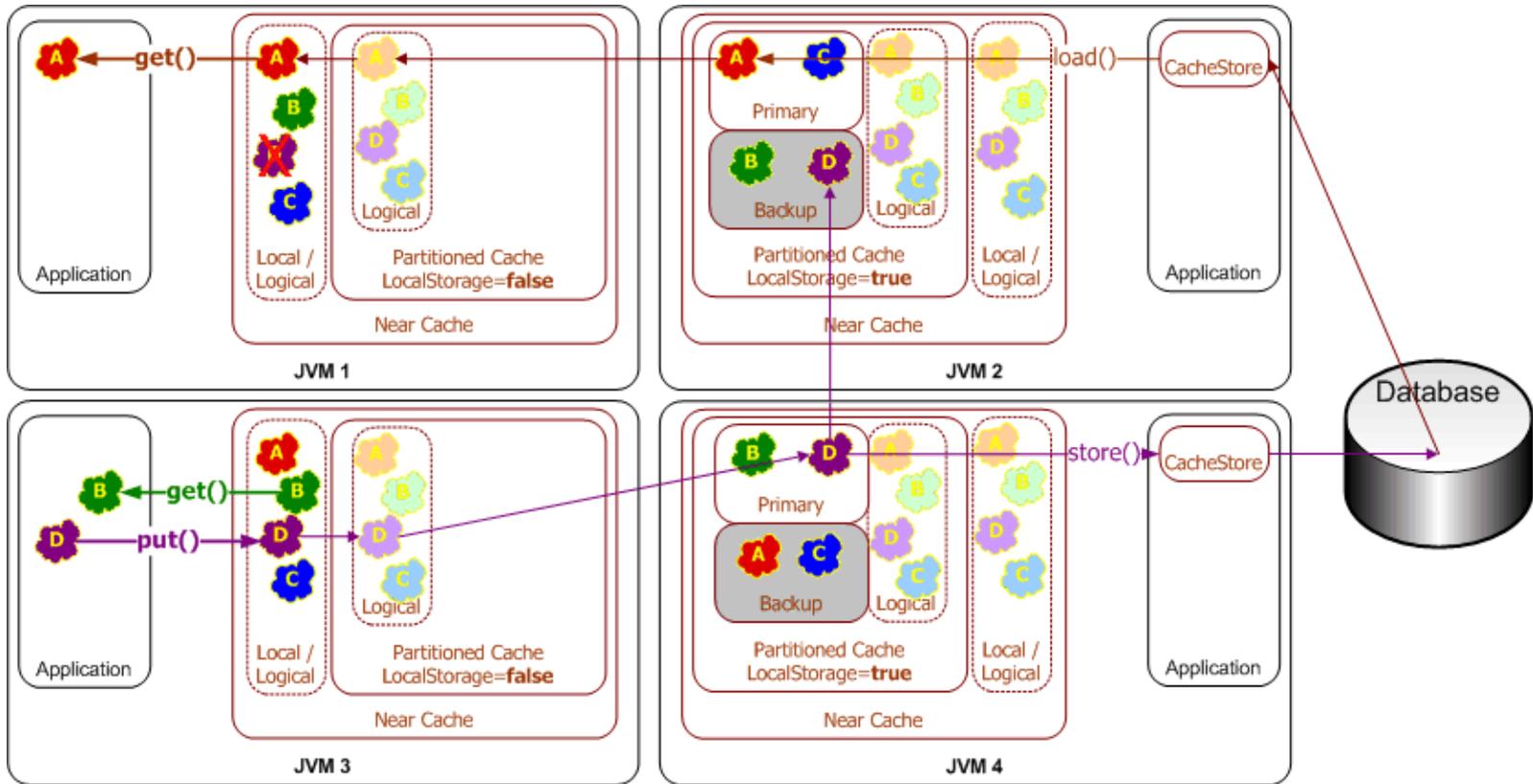
Cache-Through: Architecture

- Cache-Through places the cache between the client of the data source and the data source itself, requiring access to the data source to go through the cache
- A Cache Loader represents access to a data source; when a cache is asked for data, if it is a cache miss, then any data that it cannot provide it will attempt to load by delegating to the Cache Loader
- A Cache Store is an extension to Cache Loader that adds the set of operations generally referred to as Create, Read, Update and Delete (CRUD)

Cache-Through: Partitioning

- Cache-Through operations are always managed by the owner of the data within the cluster
- Concurrent access operations are combined by the owner, greatly reducing database load
- Since the cache is aware of updates, Write-Through keeps the cache and db in sync

Read- and Write-Through

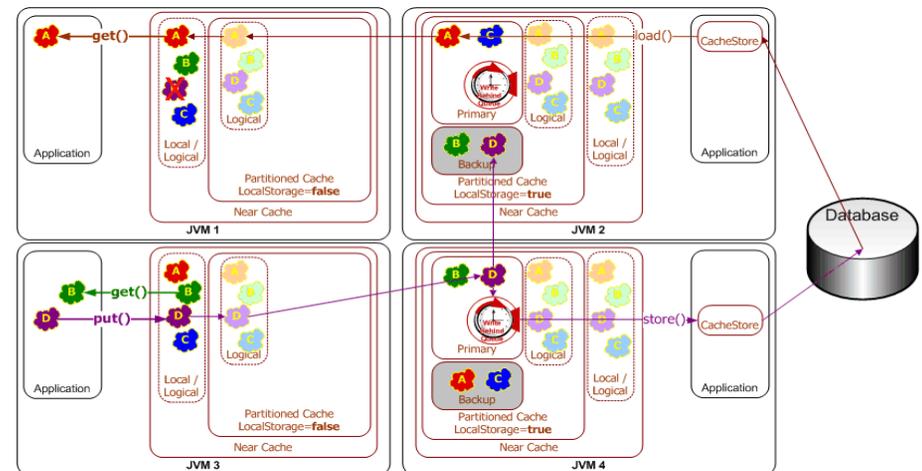


Cache-Through: Summary

- **Performance:** Reduces latency for database access by interposing a cache between the application and the data; database modifications additionally involve a cache update
- **Scalability:** Channels and combines data accesses; may significantly reduce database load
- **Uses:** Any time a cache needs to transparently load data from a database

Write-Behind: Description

- Write-Behind accepts cache modifications directly into the cache
- The modifications are then asynchronously written through the Cache Store, optionally after a specified delay
- The write-behind data is clustered, making it resilient to server failure



Write-Behind: Conclusions

- **Performance:** Low-latency for cached reads and writes
- **Scalability:** The same extreme read/write scalability of the cache, and significantly reduced load on the db
 - Coalesced write-through of multiple modifications to an object
 - Batched write-through of modifications to multiple objects
- **Uses:** When write performance is important, data source load is high, and/or when an application has to be able to continue when the data source is down
 - Only use when all writes to the data source come through the cache, and db-level auditing is not required

Topology Quiz

What cache topology would be optimal for this application?

- Caching user preferences for an in-house application
- Several hundred concurrent users
- Preferences updated a few times per day

What cache topology would be optimal for this application?

- Caching 10GB of financial portfolio data
- Read-heavy, updated nightly
- Several hundred users
- Several thousand requests per minute

What cache topology would be optimal for this application?

- Logging user interactions to a database for internal auditing purposes
- 1000 updates per minute

Lesson 1: Use an MVC Architecture

- Model/View/Controller (MVC, a.k.a. Model2)
 - Model: Domain-specific representation of the information the application displays and on which it operates
 - View: Renders the model into a form suitable for interaction, typically a user interface element or document
 - Controller: Responds to events—typically are user actions or service requests—and invokes changes on the model

Lesson 1: Use an MVC Architecture (Cont.)

- Clear delineation of responsibility; for example
 - Cache is used in the Model
 - Cache Loader/Store is the DAO
 - Cache contains the application's POJOs/Value Objects
 - View pulls data from the Model
 - Goal is to ensure that all accesses are served from cache
 - Controller affects the Model
 - Modifications via Write-Through or Write-Behind

Lesson 2: Specify the Data Access

- There are multiple ways to access data
 - ORM (JDO, EJB3, Hibernate, etc.)
 - Cache API (i.e., transparently via a Cache Loader)
 - JDBC (or other direct integration API)
- Most applications have a “best way”
 - Large-scale set oriented access usually indicates JDBC
 - Mix of set- and identity-oriented access indicates ORM
 - Identity-oriented access may indicate Cache API

Lesson 2: Specify the Data Access (Cont.)

- Picking the wrong approach is disastrous
 - RDBMS (JDBC) optimized for set-based queries and operations, including joins and aggregates, but crumbles with heavy row-level access (1+N access pattern, etc.)
 - ORMs can bog down badly on large set-based access
 - JCache API is built around identity-based access, not set-based access

Lesson 2: Specify the Data Access (Cont.)

- Not always an obvious “best choice”
 - Some applications have a mix of intensive row-level and large set-level operations, which lend themselves poorly to any single approach
 - Even a well-architected and carefully-designed application will often have a few “exceptions to the rule” that require the specified approach to Data Access to be circumvented
 - It is sometimes necessary to use different approaches for different classes of data within the same application

Lesson 2: Specify the Data Access (Cont.)

- Optimizations may be available for each
 - It is often possible to cache JDBC™ software result sets
 - Most ORMs have effective support for pluggable caches, such as Hibernate's "L2" cache support
 - Some ORMs, such as KODO Java Data Objects (JDO) API, have optimizations for set-based operations that can translate some operations directly into optimized SQL that performs the entire operation within the RDBMS
 - Caches may provide extensive query support; for example, Coherence includes parallel query with indexes and cost-based optimizations

Lesson 3: Design a Domain Model

- Domain Model includes two aspects of application modeling
 - Data Model: Describes the state that the application maintains, both in terms of persistent data (e.g., the “system of record”) and runtime data (sessions, queued events, requests, responses, etc.)
 - Behavioral Model: Describes the various actions that can affect the state of the application; very similar to the concepts behind SOA, but at a much lower level

Lesson 3: Design a Domain Model (Cont.)

- Domain Model is not dissimilar from SOA
 - Data Model: Analogous to the information encapsulated and managed behind a set of services
 - Behavioral Model: Analogous to the set of services exposed by a broker
- The concept of a Domain Model is technology-neutral; it can even exist only in the abstract
 - Modeling is a tool, not a religion

Lesson 3: Design a Domain Model (Cont.)

- Domain Model has value
 - Allows the Data Model to exist independently of the behavioral model, supporting the separation of a controller from the model in an MVC architecture
 - The behavioral model is the basis for the events that a controller is required to support
 - With an abstract data model that reflects application concerns instead technology concerns, it is much more likely that the resulting data model implementation will be more easily used by the view and the controller

Lesson 3: Design a Domain Model (Cont.)

- Domain Model is not new
 - OO developers have been using Domain Modeling for years
 - Application Developers that double as DBAs have used modeling to “get their ideas down” into a design that could provide both optimal application implementation and optimal database organization
 - SOA is the publishing of a behavior model that is intended to be publicly-accessible, with data models often directly reflected in the service request and response data

Lesson 4: Find the Natural Granularity

- Every application has a natural granularity for its data
 - Relational data models have a normalized granularity from which tables naturally emerge
 - Optimized JDBC software-based applications have a statement execution granularity and a Result Set granularity
 - ORM-based applications and cache-intensive applications often have an OO granularity that mirrors the data model
 - Caches have an identity granularity of access

Lesson 4: Find the Natural Granularity (Cont.)

- Caches will typically exist for each major class of application object
 - e.g., accounts, symbols, positions, orders, executions, etc.
- Each cache will tend to have a natural key
 - e.g., account id, symbol, account id + symbol, order id, etc.
- Application objects tend to be complex
 - Contain “owned” objects, e.g., purchase order contains lines

Lesson 5: Decouple Using Identity

- Store, Load, Provide and Manage the Identity of related model objects
- Provide accessors for related model objects by using Identity de-reference (i.e., cache access)
- Read-only models tend to have more lee-way
 - Soft references
 - Transient reference fields

Lesson 5: Decouple Using Identity (Cont.)

- Simplifies management of large object graphs
- Enables efficient lazy loading of object graphs
- Works well with...

Lesson 6: Use an Immutable Model

- From the View, the Model should be treated as if it is read-only
- From the point of view of the Controller, the model that is shared across threads should be treated as immutable, for example just in case the View is using it on a different thread

Lesson 6: Use an Immutable Model (Cont.)

- Since most applications are not read-only, the Controller does have to modify the data represented by the Model
- When the Controller needs to modify the Model, it can obtain mutable clones of the shared model, and manage them...

Lesson 7: Use Cache Transactions

- ...transactionally
- When the Controller obtains cached values within a transaction, the values are actually clones of the “master” cached values
- The Controller makes its modifications to the Model in a transactionally isolated and consistent manner

Lesson 7: Use Cache Transactions (Cont.)

- For maximum scalability, most transactions should be optimistic; just as with any optimistic transaction approach, this implies that the application must handle and/or retry transactions whose optimistic checks fail
- Cache Transactions can integrate with the container's Transaction Manager via the Java EE Connector Architecture

Lesson 8: Use Queries Wisely

- Cache Queries may be optimized, and they may even be run in parallel across a cluster, but they are probably at least an order of magnitude more expensive than identity-based operations
- If you use queries, make sure to use indexes; for example, the Coherence query optimizer uses multiple indexes on a single query, even if they don't perfectly "cover" the query

Lesson 9: Optimize Serialization

- Objects that are stored in a cache may need to be serialized, and Java technology's default object serialization is relatively inefficient
- Implementing the Externalizable interface may help slightly
- Serialization using data streams instead of object streams can make a phenomenal impact
 - ExternalizableLite interface
- Serialization performance improvements are up to an order of magnitude, and the reduction in size can be up to 80%

Lesson 9: Optimize Serialization (Cont.)

- Since Java technology does not have an object-cloning interface, classes that do not have a public clone() method may require serialization and deserialization in order to be cloned
- Since Cache Transactions may need to clone an object to create a copy within a local transaction, optimized serialization can even improve the performance of transactions

Lesson 10: Use Good Identities

- An Identity implementation must provide correct hashCode() and equals() implementations, and cache the hash-code!
- A good toString() implementation helps with debugging (and not just for Identities!)
- If feasible, make your Identity classes immutable
- An Identity must be Serializable, and its serialized form should be stable; two instances should serialize to the same binary value if and only if equals() returns true
- Java programming language's String, Integer, Long, etc. are perfect

Lesson 11: Cache in the Right Scope

- HTTP Session objects can be used for caching user- or session-specific information; don't use them as a cache for global information
- Conversely, don't use a global cache for user-specific caching when the HTTP Session would do just fine

Lesson 12: Never Assume

- Always verify that it works as expected
 - We have seen caches in production that literally were not even getting used, and we have seen caches that had not even been configured—or were badly mis-configured
 - Load test and use Java™ Management Extensions (JMX™) to monitor what caches exist, how big they are, what their hit rates are, and if the stats are as expected



the
POWER
of
JAVA™



JavaOne
FOR THE POWER OF THE POWER OF POWER

Q&A

Cameron Purdy

<code>



the
POWER
of
JAVA™

 **TANGOSOL™**



JavaOne
THE JAVAAUGUST AND JAVASEPTEMBER CONFERENCES

Distributed Caching: Essential Lessons

Cameron Purdy

President Tangosol

www.tangosol.com

TS-1402